DOTTORATO DI RICERCA IN
INGEGNERIA DELL'INFORMAZIONE
XIV CICLO

Sede Amministrativa
Università degli Studi di MODENA e REGGIO EMILIA

TESI PER IL CONSEGUIMENTO DEL TITOLO DI DOTTORE DI RICERCA

Interactive Constraint Satisfaction Problems for Artificial Vision

Relatore: **Prof. Cesare Stefanelli**

Correlatore: **Prof. Paola Mello**

CANDIDATO **Marco Gavanelli** _____

VISIT…

ACKNOWLEDGMENTS

I would like to thank all those people who made this thesis possible and an enjoyable experience for me.

In particular, I wish to thank Rita Cucchiara, Evelina Lamma, Paola Mello, Massimo Piccardi, and Cesare Stefanelli for the continuous support in all phases of this work. I also wish to thank Carmen Gervet for valuable suggestions and help during my visit in IC-Parc.

A special thank goes to Michela Milano, who has always encouraged me during my studies, guiding my work and helping whenever I was in need.

ABSTRACT

Interactive Constraint Satisfaction Problems for Artificial Vision

Marco Gavanelli , Ph.D.

Università degli Studi di Ferrara

Artificial Vision is an important field of Artificial Intelligence: the human brain is able to derive various types of information about the outer world from vision. However, the semantic interpretation of objects is a hard task for a computer. Constraints have been widely used for visual recognition, because they are able to describe very complex models and situations in a natural way [115, 86]*.

Constraint Logic Programming [67] is a new paradigm of declarative programming languages that is able to deal declaratively and efficiently with problems described with constraints.

In this thesis, we addressed problems that stem from Artificial Vision with constraints. We developed a 3D visual search system able to detect an object described with constraints in an image. We provide constraint models for dealing with the problems in Artificial Vision. We generalize the applied techniques to other important fields of Artificial Intelligence, developing new tools of Constraint Satisfaction and Optimization in Constraint Logic Programming. In order to obtain *efficiency*, we propose a tight interaction between the Constraint Solver and the provider of visual features. We give a theoretical model, solving algorithm and a language extension.

*Bracketed references placed superior to the line of text refer to the bibliography.

We model 3D objects with an object-centered model, based on visual information. From each viewpoint, only part of the visual features of a 3D object can be seen; we model objects by adding *invisible features* in the domains and select only the most informative assignments by means of a partial order.

DESCRIPTORS

Visual Search
Constraint Satisfaction
Constraint Logic Programming
Interactive Constraints
Partial Order

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

We will usually represent (unknown) variables with capital letters (e.g., X, Y) and values with small letters (e.g., a, b).

We will use the set membership superscript ($\in$) to distinguish constraints imposed on sets, with the convention that $S\ \theta^{\in} v$ stands for $\forall X \in S, X\theta v$ (where S is a set variable, θ is a binary relation and v is a value). In prefix notation, we will write $c^{\in}(S, v)$ meaning that $\forall X \in S, c(X, v)$.

$\overline{X}$	a vector.
θ	a binary relation.
$\preceq$	a Partial Order.
$\leq$	a Total Order.
$X :: D$	the association of the domain D to the variable X; declaratively equivalent to $X \in D$.
$\emptyset$ or $\{\}$	the empty set.
$\{a, b, c, d\|T\}$	the set consisting of $\{a, b, c, d\} \cup T$, where $\{a, b, c, d\} \cap T = \emptyset$.
$X \mapsto v$	variable X is assigned value v.
D_X	the domain of X.
$\widehat{AB}$	the part of a curve comprised between points A and B.
$\mathcal{P}X$	the Powerset of X, i.e., $\mathcal{P}X = \{Y s.t. Y \subseteq X\}$

ABBREVIATIONS

AC	Arc-Consistency.
B&B	Branch-and-Bound.
CHR	Constraint Handling Rules.

CLP	Constraint Logic Programming.
COP	Constraint Optimization Problem.
CSP	Constraint Satisfaction Problem.
FC	Forward Checking.
FD	Finite Domains.
GAC	Generalized Arc-Consistency.
GLB	Greatest Lower Bound.
IC	Interactive Constraint.
ICSP	Interactive Constraint Satisfaction Problem.
IFC	Interactive Forward Checking.
IMFC	Interactive Minimal Forward Checking.
KAC	Known Arc-Consistency.
LAC	Lazy Arc-Consistency.
LUB	Least Upper Bound.
MAC	Maintaining Arc-Consistency.
MFC	Minimal Forward Checking.
MOP	Multi-objective (or Multi-criteria) Optimization Problem.
PCOP	Partially-ordered Constraint Optimization Problem.

PART I

Introduction

1.0 Artificial Vision

Artificial Vision is a very exciting, yet complex, field of Artificial Intelligence. The human brain is able to process the enormous amount of information coming from the eyes at an amazing speed, and relies heavily on the gathered knowledge. We can derive huge amounts of information from an image: we are able to infer distances, position of objects, type of materials, planarity of surfaces, smoothness, and many other object characteristics.

Also, visual recognition is desirable in many practical and industrial applications. In fact, nowadays digital cameras are very cheap, and since they can provide huge amounts of data, they can be a convenient way to acquire information about the environment.

However, many recognition tasks that are naturally accomplished by our brain every second, are very hard for a computer system. We are able not only to recognize objects we are familiar with, but also objects we have never seen before. Given a set of examples, we can recognize similar objects. We are able to identify an object either given its shape, or its description, or some characteristics. We are able, for example, to *"find in an image an object with a circular shape"* or to *"count how many objects have at least five flat facets"*.

In this thesis, we studied problems related to the development of vision systems that are able to find an object in an image, matching a description. The object will be described by the user in a suitable language. We made some experiments on 2D recognition, then we designed and implemented a visual search system for 3D object recognition.

The techniques developed in the system should not be *ad hoc* solutions, but must be general and possibly applicable even if some design issues are changed. Algorithms and computational models should be given as instances, or extensions of constraint languages; they should not be stuck to a given type of image (color or black-and-white, providing bi or tridimensional information) or segmentation algorithm. The possible extensions to other research fields should be studied in detail.

A vision system must face many issues; in the following sections, we address the main problems.

1.1 Modelling

In order to obtain semantic understanding, the object must be described in a way understandable by the vision system: the system will have a *model* of the object to be recognized and an *algorithm* to perform the matching between possible models and image parts.

The model should be expressed in a language possibly easily understandable also by a human; otherwise, modelling an object would be very complex and unnatural. It should also be invariant to rotation, translation and other visual deformations; otherwise a different model must be provided for each possible view of the object. The model should be easily modifiable: if we described an object, in order to describe similar objects we do not want to start a new model from scratch. For instance, a cube is just a particular instance of parallelepipedon with square facets; once we described a parallelepipedon, we want to exploit the definition of *parallelepipedon* in the model of the cube.

We think that the Constraint description of the object is good for a series of reasons. First of all, constraints are a declarative language, that allows fast prototyping. We can naively describe an object and, if the description is not selective enough, we can add more constraints. On the other hand, if the object does not satisfy some too tight constraints, they could be easily removed, without changing the whole model. Also, if a model is given by constraints, its specializations can be simply given by adding constraints; for example, if we have defined a model $parallelepipedon(S_1, S_2, S_3, S_4, S_5, S_6)$, where S_i are the facets, we can simply add the constraints $(\forall i) square(S_i)$. Models can be grouped to constitute complex models. For instance, if an object is composed by a pyramid and a cube, we can easily combine the models *pyramid* and *cube* to have the model of the complex object.

Second, constraints are embedded in constraint systems, like Constraint Logic Programming [67], and allow efficient matching, because they embed state-of-the-art techniques.

1.2 Efficiency

The recognition task is hard, so we have issues of *efficiency*. The Constraint model allows to greatly reduce the search space, by avoiding to generate and test many impossible solutions. Although the search space is in general drastically reduced by constraint

propagation, a source of inefficiency is still present. Constraints cannot handle the low-level representation of the image, so an acquisition system is required to extract significant information from the figure. Constraints can only handle *symbolic information*, while an image is a type of *signal*. The acquisition phase is usually very expensive, as it involves deep processing of the low-level image representation.

For this reason, more interaction should be introduced in the communication protocol between the constraint solver and the acquisition module. We propose the *Interactive Constraint Satisfaction Problem* (ICSP) [19] model, where domains are partially known and *interactive constraints* are used to request only the necessary information from an acquisition system. We propose several algorithms to solve an ICSP, and the corresponding language extension belonging to the class of Constraint Logic Programming. The extension allows interaction of the system with the user, or with an acquisition system, for the request of domain values. It considers domains as first-class objects, and provides primitives to declaratively insert new values in domains. It can interface with other CLP languages on the set sort.

Again, the framework is very general, it was used in different types of applications besides visual search [5], and the algorithms were proved efficient with various types of benchmarks [20, 46].

1.3 Effectiveness

The object will be described in a way independent from the viewpoint, so the user will not have to provide a description of each possible viewpoint. A fairly complex object can have a big number of possible views; requiring the user to provide a description of each possible view is often far from practical.

On the other hand, the system should be able to infer which views are possible and which are not, in order to provide correct answers.

Describing the object and not the views means facing the problem of incomplete knowledge: in an image (thus, in a view), some facets of a 3D object will be invisible. To deal with hidden parts of the object, we proposed *visual constraints* and the Partially-ordered Constraint Optimization Problem (PCOP) model [43]. We developed corresponding algorithms and showed that the provided techniques could be used in many interesting fields, like Multi-objective Optimization and overconstrained problems.

1.4 Synopsis

In this thesis, we first provide some preliminaries, about constraints satisfaction and constraint languages (Part II). Then we show how to address the artificial vision problems with constraints (Part III). In Part IV we provide a model and algorithms for interaction in Constraint Satisfaction Problems, and in Part V we describe the corresponding Constraint Logic Programming language. In Part VI we study the problems of modelling 3D objects; we propose the model of Partially-ordered Constraint Optimization Problem and we show its usefulness in other fields apart from Visual Search. Finally, we give conclusions and suggest some future work.

PART II

Preliminaries

2.0 Constraint Satisfaction Problems

2.1 Definition

Many problems in Artificial Intelligence can be considered as instances of Constraint Satisfaction Problems.

Definition 2.1. *A* Constraint Satisfaction Problem (CSP) *is a triple* $P = \langle X, D, C \rangle$ *where* $X = \{X_1, X_2, \ldots, X_n\}$ *is a set of unknown variables,* $D = \{D_1, D_2, \ldots, D_n\}$ *is a set of domains and* $C = \{c_1, \ldots, c_m\}$ *is a set of constraints. Each* $c_l(X_{i_1}, \ldots, X_{i_k})$ *is a relation, i.e., a subset of the cartesian product of the involved variables* $D_{i_1} \times \cdots \times D_{i_k}$*. An assignment* $A = \{X_1 \mapsto d_1, \ldots, X_n \mapsto d_n\}$ *(where* $d_1 \in D_1, \ldots, d_n \in D_n$*) is a solution iff it satisfies all the constraints.*

If all the constraints in a CSP are unary or binary (involve at most two variables), we have a *binary CSP*. Each CSP can be converted into a binary CSP [4], so the algorithms developed for binary CSPs can be applied to any CSP. Binary CSPs are often represented as graphs: each variable is a node and each constraint is an arc that links the involved variables.

Applications include scheduling problems, computer graphics, natural language processing, timetable assignments, database systems, molecular biology, business applications, electrical engineering, circuit design, etc.

2.2 Algorithms

In general, the tasks posed in the constraint satisfaction problem paradigm are computationally intractable (NP-hard). Various types of algorithms have been proposed to solve Constraint Satisfaction Problems; an exhaustive survey is beyond the scope of this thesis. A good introduction is given in [111].

We can divide constraint solving algorithms in systematic search algorithms, heuristic search algorithms and consistency techniques.

2.2.1 Systematic Search Algorithms

Systematic search algorithms build a search tree and explore it to find a feasible assignment.

2.2.1.1 Backtracking. Backtracking (or Standard Backtracking, or Chronological Backtracking) [53] incrementally attempts to extend a partial solution that specifies consistent values for some of the variables, toward a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. We have a labelling phase and a test for consistency phase.

In the labelling phase an unassigned variable is chosen and assigned a value in its domain. In the test phase, the last assignment is checked for consistency with all the other variables that were assigned before. If an inconsistency is detected, the last assignment is undone, and another value is chosen.

This algorithm can be improved, considering that the backtracking point can be chosen more wisely. In other words, when an inconsistency is detected, the culprit of the failure could be a previous assignment, before the last one. The Backjumping [42] algorithm keeps memory of the last check that failed and jumps back to the culprit of the failure. Backmarking [41], instead, backtracks chronologically (undoes the last assignment) but tries to remember the causes of a failure in forward moves. For example, if the assignments $A \mapsto 1$ and $B \mapsto 2$ were found conflicting, then they will conflict again, so if we can remember the conflict during the rest of the search and avoid checking again the same couple of assignments. In the same way, successful constraint checks can be remembered as well.

2.2.1.2 Forward Checking. Forward Checking (FC) [55] is a commonly used algorithm for solving CSPs, because it is simple and effective. This algorithm interleaves a labelling phase and a propagation phase. In the labelling phase a tentative value v is assigned to a given variable X as in Standard Backtracking. In the propagation phase, unassigned variables connected to X are considered, and all values inconsistent with v removed from the corresponding domains, so only consistent values remain.

When a domain becomes empty, we have a failure, and the algorithm (chronologically) backtracks.

In order to save constraint checks, a lazy version of the Forward Checking algorithm, called Minimal Forward Checking (MFC) [23, 3], has been proposed. This algorithm minimizes the number of constraint checks by keeping only one consistent element in each domain, while the others are left unchecked. If the candidate is removed, another value must be found consistent with all the given constraints, so the next value will be checked for consistency against all the past-connected variables. On the other hand, it needs to keep track of both the checked and unchecked values. Thus more complex data structures have to be maintained. For this reason, some overhead is introduced, and the algorithm is more efficient than Forward Checking only if each constraint check is a hard task.

2.2.2 Consistency Techniques

In order to reduce the complexity of the search, some pre-processing can be performed. Domain values that cannot lead to a solution can be deleted without eliminating any feasible solution.

Definition 2.2. *A unary constraint $c(X_i)$ is* Node-Consistent *iff $\forall v \in D_i,\ v \in c(X_i)$; i.e., for each value v belonging to the domain of variable X_i is consistent.*

Definition 2.3. *A binary constraint $c(X_i, X_j)$ is* Arc-Consistent *iff $\forall v \in D_i\ \exists u \in D_j$ such that $(v, u) \in c(X_i, X_j)$ and vice-versa. In other words, for each value v belonging to the domain of variable X_i there exists a value u in the domain of X_j that is consistent with v and for each value in the domain of X_j there exists a supporting value in the domain of X_i.*

A Constraint network is Arc-Consistent if all the constraints are Arc-Consistent. Every binary CSP can be converted into an equivalent (i.e., with the same solution set) CSP which is Arc-Consistent. Many algorithms achieving Arc-Consistency have been proposed; we will cite AC-2 [115], AC-3 [80], AC-4 [84], AC-5 [61], AC-6 [6] and AC-7 [7]. All these algorithms are based on the following idea: if a binary constraint $c(X_i, X_j)$ is not arc-consistent, there will be a value v that is not supported. Supposing that v is in the domain of X_i, there is no value in the domain of X_j that is consistent with v. In this case, v can be safely removed from the domain of X_i. When we delete a value, we have a list of things that must be re-considered. In AC-3, that is often used because of its simple implementation, the list contains constraints (or, equivalently, arcs of the constraint graph). AC-3 is reported in Figure 2.1.

```
procedure REVISE(X_i, X_j)
   DELETE ← false;
   for each v in D_i do
      if there is no such u in D_j such that (u, v) is consistent,
         then
            delete v from D_i;
            DELETE ← true;
      endif;
   endfor;
   return DELETE;
end REVISE

procedure AC-3
   Q ← C   % Set of all the constraints
   while not Q empty
      select and delete any constraint c(X_k, X_m) from Q;
      if REVISE(X_k, X_m) then
         Q ← Q ∪ {c(X_i, X_k) such that c(X_i, X_k) ∈ C, i ≠ k, i ≠ m}
      endif
   endwhile
end AC-3
```

Figure 2.1 Algorithm AC-3

Other levels of consistency have been defined: Path-Consistency [85] and k-consistency [33, 34] achieve higher level of consistency, but they are more expensive. In most cases, Arc-Consistency is a good tradeoff, hence it is widely used.

If we deal with non-binary constraints, Arc-Consistency can be extended:

Definition 2.4. *An n-ary constraint* $c(X_1, \ldots, X_n)$ *is* Generalized Arc-Consistent (GAC) *iff* $\forall v \in D_i$, $\forall j \neq i$, $\exists v_{k_j} \in D_j$ *such that* $(v_{k_1}, \ldots, v_{k_{i-1}}, v_i, v_{k_{i+1}}, \ldots, v_{k_n}) \in c(X_1, \ldots, X_n)$.

Also, a lazy version of Arc-Consistency has been proposed [103]. Lazy Arc-Consistency (LAC) tries to minimize the number of constraint checks, by delaying checks on the whole domain if it is not strictly necessary. The basic observation is that Arc-Consistency algorithms are able to detect inconsistency when one domain becomes empty. So, in order to discover inconsistency, checking all the elements in the domain is not strictly necessary: if a consistent element is found, the other elements are not checked.

Consistency techniques can be easily used in a backtracking search algorithms. The backtracking algorithm that, for every node of the search tree enforces Arc-Consistency is called Maintaining Arc-Consistency (MAC) or Really Full Look Ahead.

2.2.3 Heuristic Search Algorithms

Various heuristic search algorithm have been proposed, based on various ideas or metaphors of nature processes, like hill-climbing, tabu-search [52], simulated annealing [71], neural networks, genetic algorithms [62]. We will not give a detailed description of these algorithms; the interested reader can refer, for example, to [111].

These algorithms can be very fast in finding a solution, but they are incomplete, meaning that they are not always able to find a solution, even if it exists, and that they are unable to prove unsatisfiability.

3.0 Constraint Optimization Problems

In some cases, finding a feasible solution is not enough, and a *preference* between solutions must be expressed. This preference is usually given with a function, that maps solutions into a cost (for minimization problems) or to a profit (for maximization problems).

3.1 Definition

For our needs, we define the Constraint Optimization problem as follows:

Definition 3.1. *A Constraint Optimization Problem (COP) is a couple* $Q = \langle P, g \rangle$ *such that* P *is a CSP and* $g : D_1 \times \cdots \times D_n \mapsto S_t$ *(where* $\langle S_t, \leq \rangle$ *is a totally-ordered set) is a merit function that maps each solution tuple into a value. A solution* A *of* P *is also a solution of the COP iff* $\nexists A'$ *solution of* P *such that* $g(A) < g(A')$.

We have thus a total order induced by the function f on the set of possible CSP solutions. We will consider a maximization problem, being straightforward the extension to minimization problems.

3.2 Algorithms

The usual algorithm for solving COPs is Branch-and-Bound [78].

Branch and Bound is a general search method; a possible description of Branch-and-Bound is as follows. The whole problem is divided into two or more subproblems, as in a tree-search. When we find a solution, we impose that the solution must not be worse than the current optimal solution. A sketch of the algorithm is presented in Figure 3.1.

The comparison between the current optimum and the value in each node of the search tree can be performed in different ways. In the original formulation, an Upper Bound must be computed in every node of the search tree; if the Upper Bound is worse than the current best, the branch can be pruned off, because it cannot lead to an improvement of the solution.

```
1: let CSP = ⟨X, D, C⟩, COP = (CSP, g)
2: let AP = {CSP}                          % Set of Active Problems
3: CurSol = −∞                             % Current best solution
   While AP ≠ ∅
4:            choose P ∈ AP, let P = ⟨X_p, D_p, C_p⟩
5:            if P is inconsistent
6:                  AP ← AP \ {P}
7:                  go to step 4
8:            if P has only one solution S_p
                    % Add the constraint g(X) > f(S_p)) to all the active problems
9:                  ∀Q ∈ AP, Q = ⟨X_q, D_q, C_q⟩, C_q ← C_q ∪ {g(X_q) > g(S_p)}
10:                 let CurSol ← S_p
              Else  % Branch
11:                 Generate the set CP of children of P
12:                 AP ← AP \ P ∪ CP
13:End While
14:Return CurSol
```

Figure 3.1 B&B Algorithm

4.0 Constraint Logic Programming

Constraint Logic Programming is a class of programming languages that came as a merge of two areas: Constraint Solving and Logic Programming [68].

Logic Programming is a paradigm that allows a separation between *logic* and *control* [76]. The *logic* part is responsible for correctness and describes information, given as facts and relations, which must be manipulated and combined to compute the desired result. The *control* part is responsible for efficiency, and embeds the strategies and control of the manipulations and combinations. An ideal programming methodology would be first of all concerned with *what* is the desired result; then, if necessary, with *efficiency*, or *how* to obtain the result. In Prolog, the most common logic programming language, the *logic* is given by *first order predicate logic* and the *control* is based on *SLD-resolution*.

However, one drawback of Logic Programming is that the object manipulated are uninterpreted structures and equality only holds between objects that are syntactically identical. So, for instance, $1 + 2$ is a syntactic object that cannot unify with 3.

The Constrain Logic Programming (CLP) scheme was introduced by Jaffar and Lassez [67]. It represents a class of declarative languages, CLP($\mathcal{C}$) which are parametric in the constraint domain $\mathcal{C}$. For example, $\mathcal{C}$ can be the set of real numbers, and we have the CLP($\mathcal{R}$) [70], or the Finite Domain sort, thus we have CLP(FD). Constraint Logic Programming gives an interpretation to some of the syntactic structures and replaces unification with constraint solving in the domain of the computation.

More formally, the constraint domain $\mathcal{C}$ contains the following components [69]:

- The *constraint domain signature* $\Sigma_{\mathcal{C}}$. It is a signature (i.e., a set of functions and predicate symbols) that represents the subset of the syntactic structures that are interpreted in the domain $\mathcal{C}$.
- The *class of constraints* $\mathcal{L}_{\mathcal{C}}$, which is a set of fist-order formulas.
- The *domain of computation* $\mathcal{D}_{\mathcal{C}}$, which is a Σ-structure that is the intended interpretation of the constraints. It consists of a set D and a mapping from the symbols in $\Sigma_{\mathcal{C}}$ to the set D.
- The *constraint theory*, $\mathcal{T}_{\mathcal{C}}$, which is a Σ-theory that describes the logical semantics of the constraints.

- The *solver*, $solv_{\mathcal{C}}$, which is a solver for $\mathcal{L}_{\mathcal{C}}$, i.e., a function that maps each formula to $true$, $false$ or $unknown$, indicating that the set of constraints is satisfiable, unsatisfiable or that it cannot tell.

In this thesis, we will manly refer to the FD sort and the *Set* sort. In the FD sort, the domain of the computation can be any finite set of elements. One of the most important instances is the set of integers comprised between two values $[-Max.. + Max]$: in this case the signature contains the integers, plus operations like $+, -, *, /$ and constraints like $=, <, >, \leq, \geq$ that are interpreted as in mathematics. In most CLP languages [2, 14, 24], the solver $solv_{FD}$ is based on (Generalized) Arc-Consistency.

PART III

Constraints and Artificial Vision

5.0 Introduction

In artificial vision, many different types of problems are addressed. However, most of them can be considered as different types of matching problems. In general, we have a *model* (or a set of models) of the object we want to recognize, and an image (or a set of images); the task is to find a match models/images.

There are *Image Recognition* problems, in which we search an explanation for what appears in an image. For example, we can ask the system: "What can you see in this image?", or "can you recognize any of these objects?". The system will have a database of models and will try to find a matching model for the image parts; possibly identifying each object in the image.

Visual Search problems can be, in a sense, considered the dual of Image Recognition problems: we have a model, and we ask the system to find, in an image, or in a database of images, objects satisfying the model. An example of query could be "Give me all the images containing a house", or "Count the number of red cubes in this image", or "Find if there is a triangle with three different edges"

In both cases, the association process is very complex. The image and the model are often very different in nature. The model is an abstraction, a semantic concept: a table, a bottle, etc. The image is something purely syntactic, without an associated meaning. The recognition process is the association of the meaning to the syntactic parts of the image.

In each case, deciding the type of model is crucial. It will influence the matching algorithm, i.e., how fast, selective, efficient the matching algorithm can be. It will influence the expressivity of the possible queries: if in the model we cannot state a concept, we will have no means to perform a matching consistent with the query.

In the following paragraphs, we will give some of the ideas proposed in the literature, with advantages and drawbacks.

6.0 Models and Matching

Many different approaches are present in the literature; the first distinction is the use of global or local features. With global features, the object is described as a whole: the features considered in the description language are typical of the whole object. With local features, instead, the object is described as a collection of parts, each with properties and relationships.

In the following sections, we give some examples of the two approaches.

6.1 Global Features

With global features a function associates a value to an object; if the value lies in a given range, we have a matching between model and image. Functions often used are the area, perimeter, various types of moment invariants and Fourier descriptors. The main advantage in using global features is the fast association process. The function can be usually computed very efficiently from the low-level representation of the image and the matching process consists just in checking if a value lies in a given interval.

For example, given a figure S, described as a set of pixels, the corresponding moment is simply calculated as

$$M_{i,j}(S) = \sum_{(i,j)\in S} x^j y^k$$

Given the moments of a figure, it is possible to define functions invariant with respect to translation, rotations or affine transformation [64]. However, moment invariants are usually very sensible to noise and their value changes drastically with occlusion.

Another global feature is the description of the contour. Fourier descriptors [87] consider the contour as a path in the complex plane; since it is closed, it can be considered periodical and its Fourier transform is a series. The succession of the Fourier coefficients is truncated and considered a vector in a space. This transformation is used both for the model and for the image; the matching process is simply based on a distance (e.g., the Euclidean distance) between vectors. This method is very efficient, but it has similar drawbacks to the moment invariants.

These models allow extremely efficient matching algorithms, so they are very good for real-time systems, like for aircraft identification [29]. However, they are not very expressive:

since they only consider global features, they cannot express conditions on local features, or on relations between local features. For example, we cannot express queries like: "Find the figures containing objects with at least three straight edges", or "Count how many objects in the image have at least two perpendicular segments". For this reason, local features are considered, as described in next section.

6.2 Local Features

When, in common language, we need to describe an object, we often refer to its parts, we describe them and the relations between them. For instance, we can describe a cube as a solid, composed of six facets, each of them is a square, they are orthogonal each other, etc. For this reason, it seems quite natural to allow, in the query language, description of the parts and relationships between features.

In the literature, different types of local features have been considered: line segments [115, 86], generalized cylinders [88], superquadrics [110] and geons [8]. Of course, they provide different types of matching, objects are described in different ways and they support different query languages.

Once defined the features and the relationships the system will be aware of, we need to perform a matching. Often the information about the model and the image is encoded in a graph structure [56, 74] and the matching is done with graph-matching algorithms.

All the types of graph matching problems are instances of Constraint Satisfaction Problems, and in Constraint (Logic) Programming efficient algorithms have been developed for solving the general Constraint Satisfaction Problem.

For these reasons, constraints have been applied to different types of visual problems since the very beginning of the study on constraints [115]. In next section, we will describe the constraint satisfaction framework; then, in chapter 7 we describe how to use constraints for artificial vision problems.

7.0 Constraints in Artificial Vision

Constraints are a very general methodology for modelling objects, as they can (*i*) use many different representations: they can take into account different feature types, such as surfaces, lines, generalized cylinders, etc., and different kinds of relations among features; (*ii*) they can be easily defined independently from visual deformations, like rototranslations, sketch deformations, etc.; (*iii*) they are embedded in powerful constraint languages, like Constraint Logic Programming languages, that provide both ease of representation (thanks to the intrinsic declarativity of constraints) and efficient solving (constraint propagation aimed at reducing a priori the search space); (*iv*) they are additive: if a model is not selective enough to distinguish the objects in the addressed universe, it can be easily extended by adding new constraints.

For these reasons, constraints have been used for the matching problem of artificial vision since the beginning of the study of Constraint Satisfaction. The first Arc-Consistency algorithm was proposed for interpretations of objects [115]. In line drawing interpretation, each line can be interpreted as a convex angle (and labelled with a "+" sign), a concave angle ("-") or it can be the border of an object; the border is labelled with arrows following a clockwise order (Figure 7.1). Huffman [65] found that only few combinations were possible in line drawings, as in Figure 7.2. Waltz extended the idea to line drawings with cracks and shadows [115] and realized that such analysis of scenes can be formulated as the task of solving Constraint Satisfaction Problems. He proposed an algorithm to filter impossible values from the domains (successively called Arc-Consistency [80]); the algorithm was efficient (polynomial) and could sometimes reduce the set of possible interpretations for each element down to one.

In [119] the slots in a metal object are modelled: the slots are described with the shape of the incision (the volume removed when the slot was incised). The primitive features are

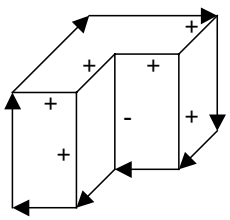

Figure 7.1 A labelled line drawing

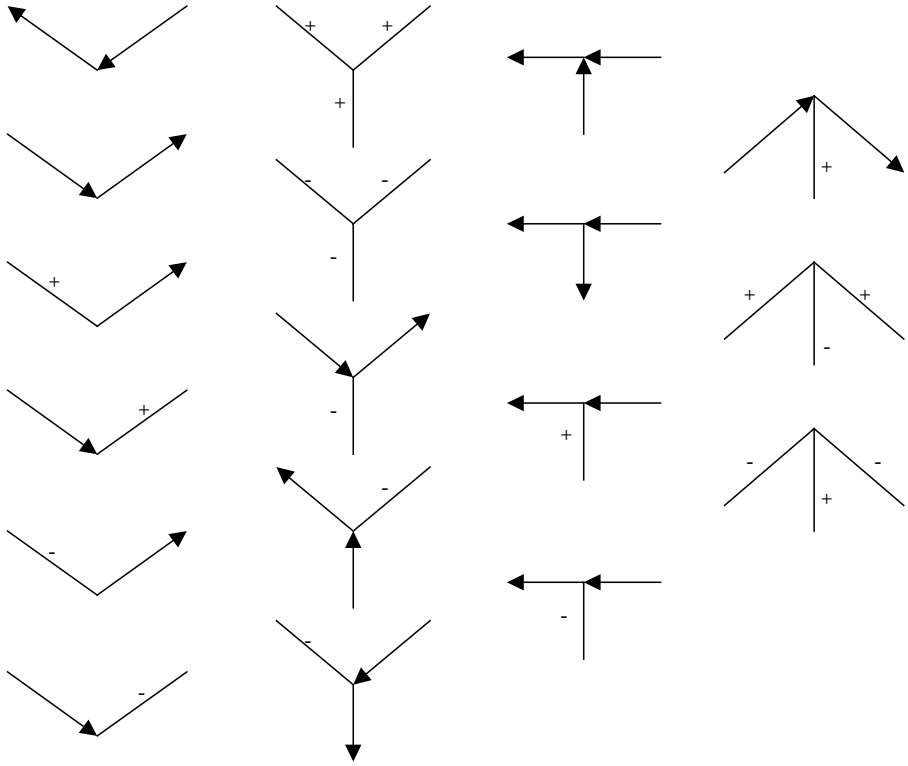

Figure 7.2 Possible combinations in line drawing labelling

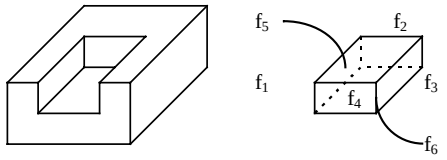

Figure 7.3 Example of a slot described with constraints

the facets of the removed volume, modelled with the normal versor exiting from the volume. The constraints are:

$Opp(f_1, f_2)$ indicates that the facets f_1 and f_2 have opposite normals,

$Perp(f_1, f_2)$ means that the two facets are perpendicular,

$Adj(f_1, f_2)$ the facets are adjacent,

$Type(f, R)$ says that the facet is real,

$Type(f, B)$ the facet is a border, i.e., it is a facet of the volume but not of the object,

$Concave(f_1, f_2)$ the facets form a concave angle,

$Convex(f_1, f_2)$ the facets form a convex angle.

These constraints are in two sets: the set of the constraints on the real surfaces and the set of the constraints involving also border surfaces. For instance, the slot in Figure 7.3 is described with constraint on real surfaces:

$$Opp(f_1, f_3) \wedge Perp(f_1, f_2) \wedge Perp(f_2, f_3) \wedge Perp(f_1, f_6) \wedge Perp(f_2, f_6) \wedge$$

$$\wedge Perp(f_3, f_6) \wedge Type(f_1, R) \wedge Type(f_2, R) \wedge Type(f_3, R) \wedge Type(f_6, R) \wedge$$

$$\wedge Adj(f_1, f_2) \wedge Adj(f_2, f_3) \wedge Adj(f_1, f_6) \wedge Adj(f_2, f_6) \wedge Adj(f_3, f_6) \wedge$$

$$\wedge Concave(f_1, f_2) \wedge Concave(f_2, f_3) \wedge Concave(f_1, f_6) \wedge Concave(f_2, f_6) \wedge Concave(f_3, f_6)$$

and constraints on border surfaces:

$$Opp(f_2, f_4) \wedge Opp(f_5, f_6) \wedge Type(f_4, B) \wedge Type(f_5, B) \wedge Adj(f_1, f_4) \wedge Adj(f_3, f_4) \wedge$$

$$\wedge Adj(f_1, f_5) \wedge Adj(f_2, f_5) \wedge Adj(f_3, f_5) \wedge Adj(f_4, f_5) \wedge Adj(f_4, f_6)$$

Constraints are able to describe very complex situations and to integrate information provided by different sources. Boshra and Zhang [12] propose a system that uses visual and tactile information. Visual features are segments and constraints can be *Same-Junction* (two segments that share and end-point), *Parallel-Edge*, *Same-Edge* (collinear segments) and *Same-Object*. A tactile sensor gives 3D information: it provides the orientation and the distance of a surface patch. Visual/Tactile constraints check the consistency of 2D and 3D information.

8.0 Modelling objects through constraints in Visual Search

In visual search applications, the model of the object should be reliable and general, in order to result possibly invariant to rotations, translation in the space, to object-camera distance and other environment factors. Thus, the object model cannot be limited to predefined absolute values, but should be based on geometric and topological relationships between features [28, 112]. Each object can be represented by means of a Constraint Graph where each part or characterizing primitive feature is modelled by a node and spatial or shape relations among object parts can be represented by arcs.

Specific aspects of a single object part may be modelled as unary constraints, such as its minimum length, color, shape, the planarity of a surface. Geometric and topological relationships between them can be represented by binary constraints (e.g., angular relationships between contours, lines or surfaces, or spatial relationships, such as *is connected to*, *touch*, *is contained*).

As a simple example, to model a rectangle, we can identify four nodes corresponding to the edges composing the rectangle (named X_1, X_2, X_3 and X_4) and impose the following constraints (the constraint name *touch* means that one ending point of the first segment must touch an ending point of the second one):

$rectangle(X_1, X_2, X_3, X_4)$:-
$touch(X_1, X_2)$, $touch(X_2, X_3)$, $touch(X_3, X_4)$, $touch(X_4, X_1)$,
$no\_touch(X_2, X_4)$, $no\_touch(X_1, X_3)$,
$same\_length(X_2, X_4)$, $same\_length(X_1, X_3)$,
$perpendicular(X_1, X_2)$, $perpendicular(X_2, X_3)$,
$perpendicular(X_3, X_4)$, $perpendicular(X_4, X_1)$,
$parallel(X_2, X_4)$, $parallel(X_1, X_3)$.

Variable domains contain the image features extracted from the scene by an image processing component. In the rectangle example, domains contain all the segments present in the scene. A rectangle is recognized when an assignment of segments to variables consistent with constraints is found.

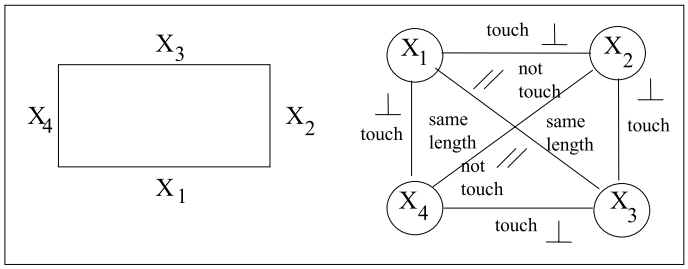

Figure 8.1 Constraint Graph of a Rectangle

Modelling objects by means of constraints allows to use constraint propagation to prune the search space, and to exploit the results coming from the constraint satisfaction and solving research field. Constraint propagation is aimed at removing the combinations of assignments which cannot lead to a consistent solution before performing the search. Therefore, a labelling strategy assigning to each variable a value (a possible segment) is intertwined with a propagation process removing combinations of assignments that cannot appear in any consistent solution, (in the example, segments which cannot form a rectangle).

Moreover, constraints are additive, so we can easily specify objects with more properties. For example, if $rectangle(X_1, X_2, X_3, X_4)$ is defined as above, to obtain the model of a square we can simply impose:

$square(X_1, X_2, X_3, X_4)$:-
$rectangle(X_1, X_2, X_3, X_4)$,
$same\_length(X_1, X_2)$,
$same\_length(X_3, X_4)$.

Finally, we can compose models to define complex objects; for example, given the model of an isosceles triangle as:

$triangle(X_1, X_2, X_3)$:-
$touch(X_1, X_2)$,
$touch(X_2, X_3)$,
$touch(X_3, X_1)$,
$same\_length(X_1, X_2)$.

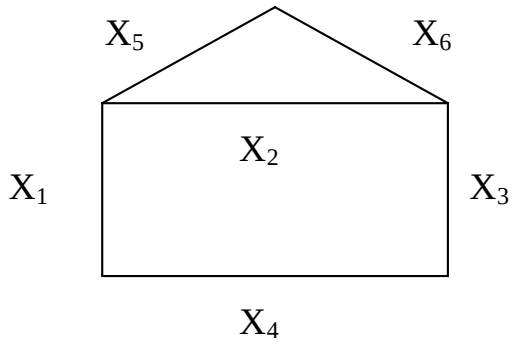

Figure 8.2 Example: Model of an envelope

in order to define the object in Figure 8.2, we can simply compose the models as follows:

$envelope(X_1, X_2, X_3, X_4, X_5, X_6)$:-
$rectangle(X_1, X_2, X_3, X_4)$,
$triangle(X_2, X_5, X_6)$.

In 3D visual search problems become more complex, because object pose must be considered. Each object can be seen from many different viewpoints; we address this problem in Part VI.

9.0 Limitations of the use of constraints for Visual Problems

Constraints are able to describe very complex situations, but they can handle only *symbolic information*, while images are a kind of *signals*.

The image itself is the result of sampling and quantization operations. It can be acquired by a camera, or scanned from a photograph. In both cases, it is usually represented as a pixel matrix, each element of the matrix contains a scalar or vector value. In grey-scale images, the value represented in each pixel is the intensity of the light in the dot. In color images, each pixel is represented by a vector, that can contain the intensity of the three primary colors (Red, Green, Blue) or values of hue, saturation, and intensity. In range images, for each pixel we have the distance of the object from the camera, thus, they provide a tree-dimensional information.

For this reason, a low level system must be used, performing the synthesis of the image features (surfaces, points, lines, generalized cylinders,...) required by the constraint model. This extraction task, called *segmentation* [63] can be very expensive, due to the difference in nature between signals and symbols.

Different segmentation algorithms have been proposed for different types of images and for different types of visual features. There are algorithms that extract the set of segments from a gray scale image and algorithms that provide the set of surfaces in a range image [63]. Also, algorithms in a same class provide different performance in terms of efficiency and reliability. In general, however, the segmentation phase is very hard, usually takes more time than the matching process. For this reason, the number of extracted features should be minimized.

Moreover, in many real-life cases information is not fully available at the beginning of the computation. For instance, consider a robot with a camera and an arm that has to grasp the objects in a box. Some blocks cannot be seen because they are under other objects. When the top objects are removed, other information becomes available to the sensors.

Unfortunately, the Constraint Satisfaction Problem (CSP), the widely used problem model employing constraints, does not deal with acquisition of the parameters, and provides no instrument to acquire data from an extraction module during the solving process. All the information possibly used during computation has to be provided in advance. Thus, data acquisition and its processing are sequentially performed, leading to an inefficient behavior of the whole system especially when the data acquisition process is computationally expensive.

The constraint solver needs all the visual features in the scene in order to create variable domains, and can start the constraint propagation process only after segmentation.

The constraint solving task can be thought as composed of two parts: the generation/acquisition of domain values and their processing. Thus, the problem solving architecture employs two components: a producer of constrained data, i.e., an image processor, and a consumer of constrained data, i.e., the constraint solver. In traditional CSPs, these two components run sequentially, since all the segments in the scene must be acquired by the vision system before constraint processing. Thus, the only information flow goes from the image processor toward the constraint solver and not vice-versa. However, feature extraction, specially in 3D visual applications, is a computationally expensive task if compared with constraint satisfaction.

We propose an alternative approach that intertwines the feature acquisition with the constraint satisfaction process thus reducing the feature extraction computational cost. The constraint solver starts the constraint satisfaction process and, as soon as some information is needed for propagating constraints or performing guesses on variable instantiation, asks the image processor to provide the needed data. The query can be simply the request of a new value, or the request of a given value satisfying certain properties. These properties can be expressed through constraints which are able to interact with the low level system and are called *Interactive Constraints*. They behave as standard constraints when information about variable domain values is known and perform acquisition when variable domains do not contain enough information.

In the next chapter, we describe the Interactive Constraint Satisfaction framework, we give some solving algorithm and we explain how to apply the framework to Visual Search.

PART IV

Interactive Constraint Satisfaction

10.0 Introduction

As we described in chapter 9, visual application problems can be considered as matching problems: image parts are recognized if they can be matched with a model known by the recognition system. A recognition process, thus, is defined by a modelling phase and a matching phase. Depending on how the model is given, different recognition strategies with different performance levels can be achieved. The model will be chosen depending on the addressed application: the model should be as simple as possible, to achieve efficiency, but must be able to distinguish the different objects in the considered world.

Constraints are a very general methodology for modelling objects, as they can (*i*) use many different representations: they can take into account different feature types, such as surfaces, lines, generalized cylinders, etc., and different kinds of relations among features; (*ii*) they can be easily defined independently from visual deformations, like rototranslations, sketch deformations, etc.; (*iii*) they are embedded in powerful constraint languages, like Constraint Logic Programming languages, that provide both ease of representation (thanks to the intrinsic declarativity of constraints) and efficient solving (constraint propagation aimed at reducing a priori the search space); (*iv*) they are additive: if a model is not selective enough to distinguish the objects in the addressed universe, it can be easily extended by adding new constraints. Several examples of constraint-based object recognition systems have been proposed [115], [86], [119], [75].

On the other hand, constraints can handle only *symbolic information*, while images are a type of *signals*. For this reason, a low level system must be used, performing the synthesis of the image features (surfaces, points, lines, generalized cylinders, ...) required by the constraint model. This extraction task can be very expensive, due to the difference in nature between signals and symbols. For this reason, the number of extracted features should be minimized.

The constraint solving task can be thought as composed of two parts: the generation/acquisition of domain values and their processing. Thus, the problem solving architecture depicted in figure 10.1, employs two components: a producer of constrained data, i.e., an image processor, and a consumer of constrained data, i.e., the constraint solver. In traditional CSPs, these two components run sequentially, since all the segments in the scene must be acquired by the vision system before constraint processing. Thus, the only information flow goes from the image processor toward the constraint solver and not vice-versa.

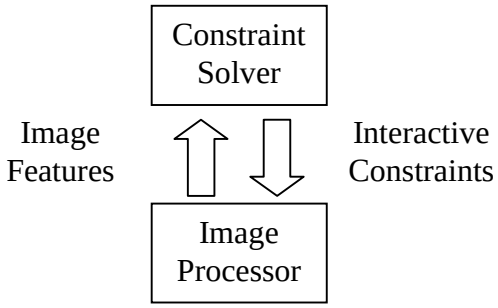

Figure 10.1 Problem solving architecture

In other words, since data acquisition and its processing are sequentially performed, we get an inefficient behavior of the whole system especially when the data acquisition process is computationally expensive. The constraint solver needs all the visual features in the scene in order to create variable domains, and can start the constraint propagation process only after segmentation.

In other types of problems, information is not fully available at the beginning of the computation. In many real-life cases, the system is not insulated, but some parameters of the constraint satisfaction problem are derived from the outer world; moreover, some parameters might be unavailable at the beginning of constraint propagation. For instance, consider a robot with a camera and an arm that has to grasp the objects in a box. Some blocks cannot be seen because they are under other objects. When the top objects are removed, other information is available to the sensors.

We state that interleaving the acquisition of domain values and their processing could greatly improve the object recognition process. Thus, we propose an extension of the CSP model, called Interactive CSP (ICSP), where domain value acquisition can be performed *on demand* only when values are needed. Furthermore, a fundamental point which can be exploited in our framework is that the acquisition process can be guided by constraints, called *Interactive Constraints*, that lead the low level component to retrieve only consistent values and to minimize useless knowledge acquisitions. They behave as standard constraints when information about variable domain values is known and perform acquisition when variable domains do not contain enough information. Interactive Constraints allow (i) to focus the attention of the feature extraction component on a restricted part of the scene, by propagating spatial and topological constraints and (ii) to constrain the feature space and assist the computation of visual features. Therefore, the constraint solver should be

able not only to prune the data set after it has been computed, but also to guide the data acquisition process.

Lazy evaluation [57] is known as a parameter evaluation mechanism which avoids a computation if its resulting value will never be used. Similarly, we avoid to consider values for constraint propagation if they are not needed. This idea has been already exploited in the field of constraint satisfaction in [23, 103] where as soon as one consistent value is found, the propagation stops in order to perform a minimal number of constraint checks.

In this chapter, we present the ICSP model and corresponding propagation algorithms. We propose a measure of the efficiency of interacting constraint satisfaction systems, and apply it to randomly-generated problems. Finally, we describe the implementation and the results on the visual search system and other applications.

11.0 The ICSP model

We now extend the CSP definition in order to allow constrained variables to range on partially or completely unknown domains.

Definition 11.1. *The* interactive domain *of a variable* X_i, $D_i = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k} | U_i\}$, *is the set of possible values which can be assigned to that variable.* $Known_i$ *is the set of values representing the known domain part* $\{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$, *and* $UnKnown_i$ *is a variable* U_i *representing the collection of not yet available values for variable* X_i. *Declaratively, the association of a domain to a variable* $X_i :: \{v_{i_1}, v_{i_2}, \ldots, v_{i_k} | U_i\}$ *holds iff*

$$X_i = v_{i_1} \vee X_i = v_{i_2} \vee \ldots \vee X_i = v_{i_k} \vee X_i :: U_i$$

Both $Known_i$ *and* $UnKnown_i$ *are possibly empty (when both are empty an inconsistency arises).*

Since the symbol "|" means set partitioning, the intersection of $Known_i$ and $UnKnown_i$, for each variable X_i, must be empty. The *interactive domain* of variable X_i can be either completely known, e.g., $D_i = \{3, 4, 5\}$, or partially known, e.g., $D_i = \{3, 4, 5 | U_i\}$, where U_i is a variable representing a collection of values which will be possibly retrieved in the future if needed, or completely unknown, e.g., $D_i = U_i$.

With no loss of generality, we define interactive binary constraints. The declarative semantics of an interactive constraint is the same as for ordinary constraints, in fact the solution set does not change.

Definition 11.2. *An* interactive binary constraint *c on variables* X_i *and* X_j, *i.e.,* $c(X_i, X_j)$, *is a subset of the cartesian product of variable interactive domains* $D_i \times D_j$ *representing sets of consistent assignments of values to variables. A binary constraint* $c(X_i, X_j)$ *is satisfied by a couple of assignments* $X_i = v_i$, $X_j = v_j$ *iff* $(v_i, v_j) \in c(X_i, X_j)$.

The strength of the interactive approach concerns the operational behaviour of interactive constraints which trigger value acquisition when not enough information is available, can guide value acquisition, and incrementally process new information without restarting a constraint propagation process from scratch each time new values are available. For instance, in a visual recognition system, constraints exploiting some form of *locality*, such as $touch(X_1, X_2)$ where X_1 and X_2 are segments, could guide the knowledge acquisition in order to let the image processing system focus only on semantically significant image parts.

Definition 11.3. *A binary* Interactive CSP *(ICSP) is a tuple* $\langle V, D, C \rangle$ *where* V *is a finite set of variables* $X_1, \ldots, X_n$*,* D *is a set of interactive domains* $D_1, \ldots, D_n$ *and* C *is a set of interactive (binary) constraints. A solution to an ICSP is an assignment of values to variables which is consistent with constraints.*

12.0 Interactive Constraint Propagation

Constraint propagation is quite different from the standard case. Consider, for the sake of clarity, only binary constraints $c(X_i, X_j)$. In the general case, both X_i's and X_j's domains contain a non empty known and unknown part. To propagate the constraint $c(X_i, X_j)$ we propagate two different kinds of constraints for each variable, each constraint is able to modify either the known or the unknown part of the domain[1]:

$$c^{\in}(K_i, X_j), \quad c^{\in}(U_i, X_j), \quad c^{\in}(K_j, X_i), \quad c^{\in}(U_j, X_i)$$

The constraint check on known parts can be performed as usual, i.e., whenever it is shown that a value cannot belong to any consistent solution, it can be deleted. In our case, however, this deletion can be performed only if the unknown part of the other domain is empty; e.g., $c^{\in}(K_i, X_j)$ can delete values from K_i only if the domain of X_j is fully known, otherwise each element in K_i could be supported by future acquisition. The check on one unknown part requires a data extraction in order to acquire new information. In addition, the data acquisition can be guided by means of interactive constraints in the sense that data acquisition retrieves values which are consistent with constraints.

Let us see a simple example considering the image in Fig. 12.1. Suppose that, after some computation, two domain variables X and Y range respectively on the domains $\{a, b\} \cup U_X$ and $\{c, d\} \cup U_Y$. The known part of the domain of X contains two segments $\{a, b\}$, while its unknown part, U_X, represents not yet available values for X. Similarly, the known and unknown part of the domains of Y are $\{c, d\}$ and U_Y respectively. A constraint between X and Y, say $touch(X, Y)$, is satisfied if and only if variables X and Y assume consistent values in their known part (e.g., $X = b$ and $Y = c$) or if the data acquisition provides consistent values for the variables.

The consistency check between $\{a, b\}$ and U_Y can call for a data acquisition that collects values for Y which are consistent with the known part of X's domain. In other words, the system collects for variable Y segments which touch a or b. This is equivalent to pose a constraint on the unknown part of Y, e.g., $touch^{\in}(U_Y, a) \vee touch^{\in}(U_Y, b)$ and guide the data acquisition by asking the low level system for values satisfying the constraints. Similarly, the constraint propagation acts on the unknown part of X and the known part of Y. Finally, the constraint between the unknown parts will check new acquired values as soon as they are available.

[1] Remember that $c^{\in}(S, X) stands for \forall K \in S, c(K, X)$, as given in the Notation (Section)

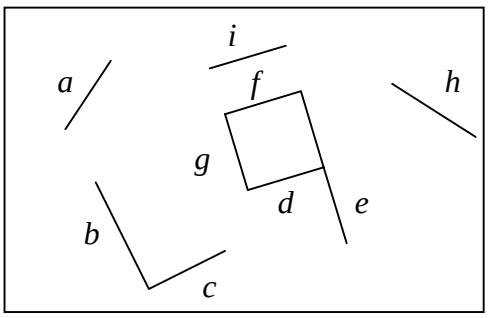

Figure 12.1 Example of 2D image

13.0 Search strategies: a framework for interaction

In this section, we define a general algorithm for performing search and propagation when data on variable domains is not known at the beginning of the constraint satisfaction process. As in CSP solving, the ICSP framework can exploit different propagation algorithms. In this chapter, we mainly refer to the Forward Checking algorithm [55] and its variants. We consider a conceptual architecture where two components interact: a constraint solver and a producer of constrained data that extracts/generates domain values and can be possibly driven by interactive constraints.

We make some hypothesis and consider some degree of freedom. The assumptions are the following:

- **End of Acquisition**: the producer of constrained data provides a set of values plus a special symbol, i.e., End Of Acquisition (EOA), when there are no more consistent values.
- **Information sources**: there can be different possible sources of information (from the constraint solver's viewpoint), with different access times. We consider a hierarchy given by the solver itself, a cache and the external world. A possible value of a variable X can be already in the (known part of the) domain D_X, or it could be in the cache, or acquired from the outer world. Another possibility is to have different information sources for different variables, e.g., different sensors.

Given these hypothesis, we have some degrees of freedom left:

- **Extraction of consistent/non consistent elements**: the acquisition of constrained data can be guided by means of constraints. The value generator/extractor can provide consistent values with constraints if it has some mechanisms for checking constraints. Otherwise, values are provided to the constraint solver that is in charge of performing the testing.
- **Number of requested values**: various acquisition strategies could be implemented. The most eager one retrieves all the consistent elements at once. The laziest one does not perform acquisition if there is at least one consistent element in the domain (i.e., the cheapest source of information). An intermediate strategy retrieves a number M of elements, where M is a parameter of the algorithm.

- **Lazy/eager constrained data acquisition**: the strategy that provides a number M of elements can be further refined; we have the following options: (i) at least M consistent values (if they exist) should be in the known part of the domain; (ii) at most M consistent values should be known. In case (i) if only $N < M$ values are available in the information source we are considering, then the system retrieves the N values from the (low-cost) source and requests $M - N$ values to the following (thus, more costly) information source. In case (ii) it retrieves N values from the (low-cost source) and does not acquire values from more costly sources. Of course, in both cases if no value can be retrieved from any information source, a failure occurs.

13.1 The Algorithm

The algorithm in Figures 13.1 and 13.2 is described in pseudo-Prolog, for what concerns the constraint propagation, and in pseudo-Pascal for the communication part. We suppose that the domain of a variable X (in particular its known and unknown part, respectively K_X and U_X) is accessible starting from the variable itself.

The algorithm chooses the above-mentioned degree of freedom in the following way. First, the value generator is supposed to extract elements that are consistent with respect to the constraints used for guiding the acquisition. Thus, the constraint solver does not need to check acquired values against these constraints. Second, the algorithm performs an eager form of acquisition, in the sense that if the acquisition requires M elements, it will retrieve either M elements if they exist, despite of where they can be retrieved, or $N < M$ if no more than N values are available in any information source. Third, the number of requested values is an algorithm parameter which can be tuned on the specific application.

As usual, constraint programs start by defining the CSP (Figure 13.1); however, an ICSP solving algorithm does not need domains to be defined at the beginning; only variables and constraints have to be given.

After the selection of the current variable (by means of some heuristics), the proposed algorithm is divided in two steps. A *labelling move* is used to find an instantiation for the current variable, and a *propagation/acquisition* step is used to remove (already acquired) values inconsistent with the current labelling, or to acquire new consistent values.

```
solve_icsp(Vars,M):- impose_constraints(Vars), ifc(Vars,M).

ifc([],_).
ifc(Vars,M) :- not empty(Vars),
      select_var(X,Vars,Future),
      i_label(X),
      i_propagation_acquisition(X,Future,M),
      ifc(Future,M).

i_label(X) :- empty(K_X),
      collect_constraints(X,C),
      acquire_values(X,C,1),
      i_label(X).
i_label(X) :- nonempty(K_X),
      select V ∈ K_X,
      assign(X,V),
      (var(X) -> i_label(X)).
      % On failure, assign/2 unbinds the variable and removes value V

assign(X,X).
assign(X,V) :-
      remove(X,V).

remove(X,V) :-
      remove V from K_X,
      impose constraint U_X ≠^∈ V.
```

Figure 13.1 The search algorithm: Labelling

```
i_propagation_acquisition(X,[],_).
i_propagation_acquisition(X,[Y|T],M) :-
      remove inconsistent values from K_Y,
      find_candidate(Y,M),
      i_propagation_acquisition(X,T,M).

find_candidate(Y,M) :-
      nonempty(K_Y), |K_Y| ≥ M.
find_candidate(Y,M) :-
      nonempty(K_Y), empty(U_Y).
find_candidate(Y,M) :-
      |K_Y| < M,
      collect_constraints(Y,C)
      acquire_values(Y,C,|K_Y| − M).

acquire_values(X,Cst,M) :-
      cache(X,Cst,M',V_c),
      extract(X,Cst,M-M',V_a),
      nonempty(V_c ∪ V_a),
      U_X = V_c ∪ V_a.

function extract(Y,Cst,M,V_a):boolean
begin
      send (Cst, M) to extractor;
      receive from extractor (Values);
      if Values ∋ EOA
          then V_a = Values \ {EOA}
          else V_a = (Values \ {EOA}) ∪ U^New
end
```

Figure 13.2 The search algorithm: Propagation

The labelling step, called `i_label`, takes as input the variable X to be instantiated, and either selects a value V in its known domain part if it exists, or retrieves one value V for X consistent with the unary constraints on X. `i_label` then assigns $X = V$.

The propagation/acquisition step (Figure 13.2), predicate `i_propagation_acquisition`, takes as input the current variable and the set of future connected variables. For each future connected variable Y, a classical FC propagation step is performed in order to remove inconsistent values if there are known values in the domain of Y. Otherwise, an acquisition is performed in order to retrieve M values for Y consistent with v (predicate `acquire_values`). Thus, the search algorithm is parametric with respect to the number of retrieved values, M, and enables the achievement of different levels of uncoupling between the constraint solver and the producer of constrained data.

`acquire_values` searches for consistent elements in a global cache memory, then, if not enough values have been retrieved, sends a set of constraints to the extraction component and waits for the acquisition. The set of elements consistent with the constraints is then declaratively assigned to the unknown domain part. If the set of retrieved values contains the EOA symbol, the domain can be *closed*: the unknown domain part becomes empty and all retrieved elements are put in the domain. Otherwise, if EOA is not encountered, the domain should be left *open* and the new unknown part is a fresh new variable U^{New}.

13.2 Specializations

As mentioned, we have considered the number M of requested elements as a parameter of the algorithm. We have three possible alternatives obtained by setting the parameter M: (i) an eager acquisition aimed at retrieving **all** values consistent with v; (ii) a lazy acquisition aimed at retrieving **one** value consistent with v; (iii) an intermediate form of acquisition which requests M values consistent with v.

13.2.1 Eager Acquisition Strategy

In case (i), we have an algorithm called Interactive Forward Checking (IFC) [20]. One of the well known and widely accepted propagation algorithms for solving CSPs is the forward checking (FC) technique [59]. The FC algorithm intertwines a *labelling* step, where a variable X is instantiated to a value v in its domain, and a *propagation* step where domain

variables linked with X by means of constraints are checked in order to remove values which are not compatible with v.

In our framework, we have to cope with partially known domains. Therefore, the operational behaviour of the FC algorithm should be changed accordingly. Intuitively, the first *labelling* step instantiates a variable X to a value v belonging to the known part of the domain if any. Otherwise, a data acquisition is performed retrieving a value v which is successively assigned to X. The *propagation* step considers domain variables $X_1, \ldots, X_k$ linked with X by means of constraints. This step removes from the known part of $X_1, \ldots, X_k$ domain inconsistent values, and (eventually) retrieves consistent data for the unknown part. Thus, the general propagation described in the previous section is specialized in the Interactive FC algorithm, simply by performing acquisition only when one of the involved variables is assigned a value and acquiring all the consistent elements for the other variable. Note that in this case, at each step of the computation, each domain can be either completely known or completely unknown. In fact, at the beginning all the domains are completely unknown, and, after an acquisition, all consistent values are retrieved, thus making the domain fully known.

13.2.2 Lazy Acquisition Strategy

In the second case, we have a lazy acquisition of domain values and the acquisition stops as soon as one consistent value has been found. The resulting algorithm is the interactive version of Minimal Forward Checking [23], and is called Interactive Minimal Forward Checking (IMFC) [20]. The algorithm maintains only one consistent value in the domain of each future variable, suspending forward checks until other values are required by the search, i.e., when the only consistent value is deleted. After the acquisition, the domain of variable X_j can be possibly still partially unknown, i.e., it is still *open*. In particular, it is composed of the retrieved consistent value v_1 and a variable U'_j representing future acquisitions, i.e., $X_j :: \{v_1 | U'_j\}$.

In order to process new data without starting the constraint propagation process from scratch each time new values are available, we impose auxiliary constraints on the new variable representing the unknown domain part. These constraints will be used if the retrieved value is deleted or a failure occurs. We will use the set membership superscript ($\in$) to distinguish constraints imposed on sets, with the convention that $S\ \theta^{\in} v$ stands for $\forall X \in S, X \theta v$ (where S is a set variable, for instance the unknown part of a domain, θ

is a binary relation and v is a value). Thus, given the constraint $c(X_i, X_j)$, we impose on the unknown domain part the auxiliary constraints $U_j \neq^{\in} v_1$ and $c^{\in}(X_i, U_j)$. As an example, consider the constraint $X_1 < X_2$, where variable X_1 is instantiated to 3 and the domain of variable X_2 is completely unknown ($X_2 :: U_2$). When the constraint is checked, a knowledge acquisition is performed for X_2. Suppose that value 5 is retrieved. Variable U_2 is declaratively assigned to the set $\{5|U_2'\}$, where U_2' is a fresh new variable. The domain of X_2 becomes $X_2 :: \{5|U_2'\}$ and on U_2' a set of constraints is imposed, stating that $U_2' \neq^{\in} 5$ and $U_2' >^{\in} 3$. These constraints are taken into account later on if value 5 is removed from the domain of X_2 during constraint propagation or upon backtracking. In the example, suppose that X_2 is assigned the trial value 5. If this choice leads to a failure, the domain is restored (i.e., $X_2 :: \{5|U_2'\}$), 5 can be removed (i.e., $X_2 :: U_2'$) and a new value can be acquired exploiting the constraints imposed on U_2'. If, after backtracking, also X_1 is unlabeled, the constraint $U_2' >^{\in} 3$ is removed.

13.2.3 An Intermediate Acquisition Strategy

Case (*iii*) considers intermediate acquisition forms. The resulting algorithm is a generalization of IMFC since, after the acquisition, the domains of the variables can still be *open* (i.e., in case the symbol EOA is not encountered).

Intuitively, the interesting case here happens when the parameter M is not ground. It will be bounded to the number of elements already present in the less expensive information source, e.g., a cache (in this case, the acquisition phase should be slightly modified as in Figure 13.3). If no element is available in that information source, it can ask for more elements from a different (slower) information source in a synchronous or asynchronous communication mode.

The intermediate approach represents a way of uncoupling the producer and the consumer of domain values. In fact, in the previous cases, the constraint solver asks for one or all consistent values, waits for the answer of the producer of domain values, and continues the computation, in a synchronous communication model. However, a more efficient approach, used in pre-fetching activities for hierarchical memories [58], enables the producer to start its computation independently. In case of multiple producers, they can have different throughput, and they can start their computation in parallel, storing the results in a common shared memory or in a communication stream. The shared memory can be also

```
find_candidate(Y,M) :-
      nonempty(K_Y).
find_candidate(Y,M) :-
      empty(K_Y), nonempty(U_Y),
      collect_constraints(Y,C)
      acquire_values(Y,C,M).

acquire_values(X,Cst,M) :-
      cache(X,Cst,M,K_X^New),
      nonempty(K_X^New),
      U_X = K_X^New ∪ U_X^New.
acquire_values(X,Cst,M) :-
      extract(X,Cst,M,V_a),
      M = |V_a|;
      nonempty(V_a),
      U_X = V_a.
```

Figure 13.3 The acquisition phase when M is unbound

considered as a form of caching, and is part of the hierarchy of information sources described earlier. When the constraint solver needs data, it retrieves from the shared memory available data, i.e., M values consistent with the interactive constraint. If the parameter M is left unbound, then the solver can retrieve from the cheapest information source all the available consistent elements, without waiting for more elements to be retrieved, thus allowing an asynchronous communication model and a higher parallelization degree. If no consistent elements are found in cache, then the solver can request more elements, then suspend the acquisition activity and continue other activities (i.e., consistency checking or acquisition of domain elements for the other variables). In this way, if the constraint solver deals with more acquisition modules, with different speeds, each retrieving values for a different variable, we could have an eager processing of fast-obtained values and a lazy processing of hard-to-compute values.

For example, consider a visual recognition system: some variables range over the segments in an image, others over surface patches. Segmentation of lines is easier than retrieval of surface patches, so in the same interval when only one surface is retrieved, all the segments in the image could be retrieved. The solver acquires then the available surfaces (i.e., only the first) thus managing lazily the domain of the corresponding variable. Then it

acquires all the segments in the image, thus manipulating the domain of the variable in an eager fashion. At this point, it can perform a more complete propagation and eventually find a domain wipe out earlier, thus increasing efficiency.

In general, it is not always possible to retrieve consistent values from the constrained data producer. In this case, the producer cannot perform the requested constraint checks and the consistency check should be performed by the constraint solver. The uncoupling and the parallel execution of the two components has the drawback of reducing the possibility of guiding the acquisition through interactive constraints. However, a form of guidance can be achieved in the same way as pre-fetching data in cache memories exploit some form of spatial and temporal locality of memory pages [58]. In fact, interactive constraints can be exploited by the producer(s) in order to retrieve information which is *potentially consistent* with constraints.

The described algorithm has one drawback; after backtracking, some of the performed computation can be redone. In fact, as stated in [23], some kind of memory must be kept of successful and unsuccessful constraint checks in order to minimize the number of checks. In our application, the minimization of constraint checks is not a central issue, because the acquisition phase is much more time consuming than constraint solving. The proposed algorithm does not redo acquisitions, because the acquired values are cached. If an application needs to minimize the number of constraint checks as well, the algorithm can be implemented with the backtracking mechanism described in section 14 [48].

13.3 Examples

13.3.1 Example: Arithmetic Constraints

In this section, we present an example of ICSP computation with numeric constraints. Arithmetic constraints have been chosen because they are very intuitive, even if the ICSP framework is better suited for problems in which domains cannot be easily computed. Consider a problem of four variables linked by the following constraints: $B \ mod \ A = 1$, $B \ mod \ D = 3$, $D \ mod \ C = 1$ and $B \ mod \ C = 4$.

All variables start with an unknown domain; suppose that the values the acquisition module retrieves are $\{5..30\}$ and the variable selection heuristics chooses the variables in lexicographic order. The behaviour of the Interactive Forward Checking algorithm is shown in Table 13.1. It is worth noting that after each propagation/acquisition step the domains

Phase	**A**	**B**	**C**	**D**
Labelling	5	$Unknown_B$	$Unknown_C$	$Unknown_D$
Acquisition	5	$\{6, 11, 16, 21, 26\}$	$Unknown_C$	$Unknown_D$
Labelling	5	6	$Unknown_C$	$Unknown_D$
Acquisition	5	6	$\{\}$	$Unknown_D$
Labelling	5	11	$Unknown_C$	$Unknown_D$
Acquisition	5	11	$\{7\}$	$\{4, 8\}$
Labelling	5	11	7	$\{4, 8\}$
Propagation	5	11	7	$\{8\}$
Labelling	5	11	7	8

Table 13.1 Example of IFC computation

are in the same status as after a classical Forward Checking propagation. However, 26 values are necessary for a CSP computation (all the values from 5 to 30), while only 9 are used in IFC propagation (i.e., 4, 5, 6, 7, 8, 11, 16, 21 and 26).

Phase	**A**	**B**	**C**	**D**
Labelling	5	$Unknown_B$	$Unknown_C$	$Unknown_D$
Acquisition	5	$\{6 \mid Unknown'_B\}$	$Unknown_C$	$Unknown_D$
Labelling	5	6	$Unknown_C$	$Unknown_D$
Acquisition	5	6	$\{\}$	$Unknown_D$
Acquisition	5	$\{11 \mid Unknown''_B\}$	$Unknown_C$	$Unknown_D$
Labelling	5	11	$Unknown_C$	$Unknown_D$
Acquisition	5	11	$\{7 \mid Unknown'_C\}$	$\{4 \mid Unknown'_D\}$
Labelling	5	11	7	$\{4 \mid Unknown'_D\}$
Propagation	5	11	7	$Unknown'_D$
Acquisition	5	11	7	$\{8 \mid Unknown"_D\}$
Labelling	5	11	7	8

Table 13.2 Example of IMFC computation

Let us consider how the IMFC algorithm solves the same example. As we can see from Table 13.2, the number of acquired values is further reduced to 6 (only values 4, 5, 6, 7, 8, 11 are retrieved); however, more labelling and propagation/acquisition steps have been performed. In addition, we introduce an overhead due to the maintenance of additional constraints on unknown domain parts. This means that this algorithm is more suitable if the acquisition process is computationally more expensive than the constraint solving part; otherwise the IFC algorithm could be preferable. Constraints on unknown domain parts, however, are powerful tools in backtracking. Consider Table 13.2, when value 5 is assigned to variable A, one value, i.e., 6, is retrieved for variable B, and on $Unknown'_B$

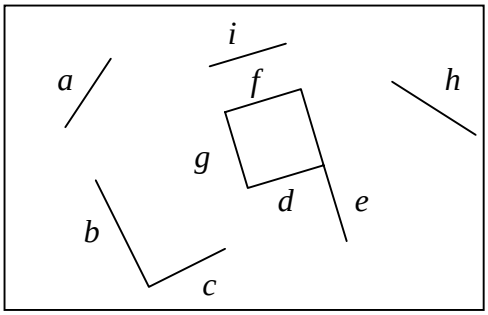

Figure 13.4 Example of 2D image

two constraints are imposed, i.e., $Unknown'_B \neq 6$ and $Unknown'_B \ mod \ 5 = 1$. When the computation fails for variable C, another value for B should be retrieved consistent with constraints on $Unknown'_B$. Value 11 satisfies the constraints and is considered for B.

13.3.2 Example: 2D visual search

Let us consider an example in the field of 2D shape recognition. The task is to recognize the first rectangle, modelled as in Section 8:

$touch(X_1, X_2)$, $touch(X_2, X_3)$, $touch(X_3, X_4)$, $touch(X_4, X_1)$, $no\_touch(X_2, X_4)$, $no\_touch(X_1, X_3)$, $same\_length(X_2, X_4)$, $same\_length(X_1, X_3)$, $perpendicular(X_1, X_2)$, $perpendicular(X_2, X_3)$, $perpendicular(X_3, X_4)$, $perpendicular(X_4, X_1)$, $parallel(X_2, X_4)$, $parallel(X_1, X_3)$.

in the scene in Fig. 13.4 by its four edges X_1, X_2, X_3 and X_4 defined as CSP variables. Variable domains are segments retrieved from the image. Initially, the variable domains are completely unknown:

$$X_1 :: U_1, \quad X_2 :: U_2, \quad X_3 :: U_3, \quad X_4 :: U_4.$$

The FC algorithm starts with a labelling step on variable X_1. Since the domain of X_1 does not contain any known value, the labelling step performs a feature acquisition process (possibly guided by unary constraints on X_1). Suppose that segment f in the image is retrieved and assigned to X_1, i.e., $X_1 = f$. Now, the FC algorithm collects all the variables linked to X_1 by means of constraints and removes from the known part of their domains all

values which are inconsistent with f. If variable domains are completely unknown, a data acquisition is performed in order to retrieve values consistent with f. Consider the two constraints stating that variables X_2 and X_4 should touch segment X_1, i.e., $touch(X_1, X_2)$, $touch(X_1, X_4)$. The constraint check results in a data acquisition process since both X_2 and X_4 domains are still unknown. Note that the feature acquisition process can be guided by the two above mentioned constraints exploiting the locality criteria embedded in the constraint *touch*. Therefore, the underlying visual system looks for one or all segments touching f thus focusing attention around f. The system collects two segments g and l[1] which are put in the domains of X_2 and X_4. The unknown parts of X_2 and X_4 are now completely known and the domains of X_2 and X_4 contain all values which touch (and thus are consistent with) f. After propagation of non interactive constraints, a labelling step starts for X_2 thus assigning l to X_2 and a second constraint propagation process starts by considering all variables involved in a constraint with X_2. In particular, propagation of the interactive constraint $touch(X_2, X_3)$ results in a feature acquisition, collecting all values that touch X_2, i.e., d and e. Note that, again, feature acquisition is focussed around l and does not need to consider the whole image. These segments are put in the (known part of the) domain of X_3 while the unknown part is deleted. The propagation step between X_2 and X_4 is the usual forward checking constraint propagation since the domain of X_4 is completely known. The FC algorithm continues the labelling and the constraint propagation process as usual since all the domains now are known.

The purpose of the interactive framework is to force the low level system to extract a number of segments which is significantly smaller than those retrieved by a non-interactive system which first collects all segments in an image and then starts the constraint propagation process. The number of extractions must be kept as small as possible, since the extraction process is the most expensive task: usually the CSP solving process employs a negligible computation time with respect to the feature extraction time. In order to avoid re-extraction of formerly acquired values (due, e.g., to backtracking), each extracted element is stored in memory after interaction; in successive computation, every segment will first be searched for in memory and then requested to the low level system. Note that the amount of memory required for this structure is only proportional to the number of elements in a domain, which is negligible with respect to the structures needed by the FC algorithm [55].

[1]Note that constraints describing the rectangle are symmetric. Symmetries should be avoided in a constraint satisfaction procedure [90]. Therefore, in practice, we do not put all the acquired segments in both domains. In the example, however, for the sake of simplicity, we omit the treatment of symmetries.

14.0 Backtracking and Lazy Domain Evaluation

As we suggested earlier in paragraph 13.2.3, our vision system does not need to minimize the number of constraint checks, because the constraint satisfaction is much more efficient than the feature extraction process. However, in some cases, minimizing the number of constraint checks could be very useful. We wanted the system to be very general, and easily modifiable to address other types of images and constraints. Some constraints could be checked only by a low-level system: the planarity of a surface, or the contact between surfaces can be checked only by a low-level system, and could be very expensive. In such cases, some form of remembering of constraint checks should be kept.

In order to exploit the full power of lazy evaluation, the chronological backtracking of CLP languages should be modified. In chronological backtracking, used, e.g., in Prolog, all the performed computation is lost, nothing is remembered or learnt about the problem. On the other hand, some search algorithms use a form of remembering of results obtained in failing branches [41, 23]. Implementing these algorithms in a language exploiting Prolog backtracking makes both complex and inefficient programs.

For this reason, we propose a backtracking rule providing the possibility to remember information from failures. In particular, this is useful if we postpone some of the calculus and perform it only when required by the search. In other words, it is suitable when we exploit lazy domain evaluation, such as in Minimal Forward Checking (MFC)[23].

In next sections, we define a new predicate that allows recording dependency information of results on choices. We show a CLP implementation of MFC exploiting the new defined predicate. Then we show a simple Prolog meta-interpreter providing the discussed predicate as a built-in and some experimental results.

14.1 A New Predicate

In CSP solving techniques, a variable is associated with a domain, which can be modified during computation. Typically, as propagation is performed, new information is inferred and domains are reduced. In other words, at the beginning every variable domain contains all the possible elements; then values are deleted thanks to constraint propagation and knowledge about the final value the variable will assume is refined.

When propagation can infer no more information about the problem, we are forced to perform a guess. Many search algorithm are based on the idea to assign a trial value to a variable and then propagate the consequences of the choice. If the choice leads to a failure, we have to undo the assignment and all the computation results that depend on it. In many important search algorithms (e.g., Forward Checking (FC) and Looking Ahead [55], Maintaining Arc Consistency (MAC) [95]), every elimination occurs because of the *last* assigned value. In other words, in these algorithms a value cannot be deleted from a domain because of a previously-made assignment and all the computation is performed in chronological order. For this reason, the chronological backtracking of Prolog is perfectly suited to implement these algorithms. However, if we want to implement an algorithm that learns during computation, we are forced to use extralogical predicates.

An idea that has been proven effective for CSP solving is *lazy domain evaluation* [121, 23, 103]. The idea behind it is that every constraint check should be performed as late as possible and only if effectively required by the search. For example, the Minimal Forward Checking (MFC) [23] algorithm is a lazy version of FC that keeps only *one* consistent value in each variable domain. If the only consistent value is deleted from the domain, another value must be selected and checked for consistency with respect to all the past connected variables. This means that a constraint involving a past connected variable can delete a value from a domain; this elimination should not be undone if the past connected variable's assignment is not undone.

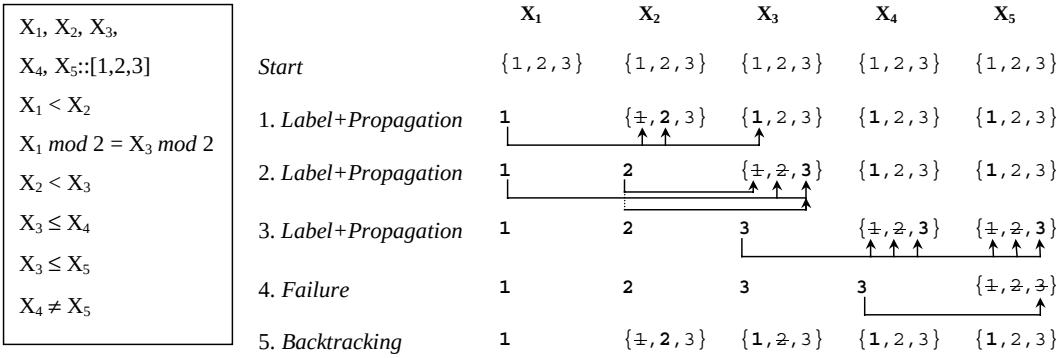

Figure 14.1 Example of Minimal Forward Checking computation

For example, consider the CSP in Figure 14.1: $X_1 < X_2$, $X_1 \ mod \ 2 = X_3 \ mod \ 2$, $X_2 < X_3$, $X_3 \leq X_4$, $X_3 \leq X_5$, $X_4 \neq X_5$ where all the variables range on the set $\{1, 2, 3\}$. Suppose X_1 is assigned value 1, then propagation starts (step 1). Value 1 is deleted from X_2's domain, value 2 is consistent and becomes the candidate (bold in the figure). Value 1

in the domain of X_3 is consistent and another labelling phase starts (step 2). X_2 is assigned value 2; the candidate of X_3 is inconsistent, so it is eliminated and the successive value 2 must be checked against the past variable X_1. This value is inconsistent and is removed; value 3 is consistent with all the constraints and becomes the candidate. Note that value 2 has been eliminated because of the value assigned to X_1 when the current variable (the last assigned variable) was X_2. This means that the value 2 must not be re-inserted in X_3's domain while the variable X_1 (the cause of the elimination) keeps its value. In particular, it must not be re-inserted when variable X_2 is unlabelled.

In the example of Figure 14.1, we have a failure and we have to unlabel variable X_2. Chronological backtracking suggests to go back to step 1, i.e. to re-insert all the domain values that were removed *after* the labelling of variable X_2. MFC, instead, goes to the status 5, i.e. re-inserts all the domain values that were removed *because of* the labelling of variable X_2: since value 2 in $dom(X_3)$ was removed because of X_1, it is not re-inserted.

It's worth noting that if we had checked variable X_1 with all the values in X_3's domain, we would not have this problem. Postponing evaluation and performing it only 'on demand' requires a non chronological backtracking rule.

What we propose is quite different from the intelligent backtracking widely studied in the past [13, 18, 15, 77]. All those proposals suggest to find a good backtracking point, undo all the computation performed after this point and then jump to it, just as if all the computation performed in this while had not been done. Instead, we do not focus on the backtracking point but on the operations performed after. When we backtrack to a certain point, not all the work has to be undone. If we undo a choice (and we are going to make another choice), only the computation depending on this choice has to be undone; all the results we got depending on previously made choices must be kept. For this reason, the system needs to know which result depends on which choice. When dealing with constraint satisfaction, a choice is simply an instantiation of a variable to a domain value; a result is the removal of an element from a domain. We suggest thus to consider a predicate `set_dependency(A,Var,Val)` with the intended meaning that the domain value `Val` must be considered removed from the domain of variable `Var` until the assignment of variable `A` is undone. This predicate can be used to implement constraints and have them delete values. Let us consider now how this predicate can be used to implement the MFC algorithm.

```
label_mfc([]).
label_mfc([Xa |T]) :-
                    instantiate(Xa),
                    label_mfc(T).

instantiate(Xa) :-  Assign the first value in the domain.
instantiate(Xa) :-  delete the first element from the domain,
                    instantiate(Xa).
```

Figure 14.2 A MFC Prolog Implementation - labelling phase

14.2 A CLP Implementation of Minimal Forward Checking

In this section we show a simple implementation of the MFC algorithm [23, 3]. The algorithm alternates a labelling phase and a propagation phase. The search starts with a call to `label_mfc`, (figure 14.2) which takes as input the list of problem variables. This predicate selects a variable (in our example, the first) tries to instantiate it, propagates constraints, then tries to instantiate the following variables. To instantiate a variable, first we try the first domain value, that is the only element checked against all the past connected variables. If we get a failure, the candidate is deleted, triggering propagation, so another candidate is found.

The propagation phase is performed by the successive activation of suspended constraints. As usually happens in CLP, constraints are activated when elements are removed from a domain; in particular, when the candidate is deleted. In Figure 14.3 the implementation of a lazy constraint is shown. The constraints are suspended waiting for one variable to become instantiated. When a constraint is activated, the first element in each future domain is checked for consistency with respect to the current assignment. If the candidate is consistent, nothing happens: the constraint just suspends in order to check successive values if the candidate is removed. If the candidate is inconsistent, it is removed (thus awaking the past constraints) and the dependency on the current assignment is recorded. Then, another candidate must be found, and the constraint performs a recursive call. As explained in [23], a recording of the removed values must be kept in order not to re-execute constraint checks. Thanks to this recording (provided by the `set_dependency` predicate), the domain element will not be re-inserted during backtracking, unless the cause of the elimination is unassigned.

```
lazy_constraint(X_a,X_f) :-
        var(X_a), var(X_f), !, suspend.
lazy_constraint(X_a,X_f) :- % Suppose X_a is instantiated
        Let E be the first element of X_f's domain
        (is_consistent(X_a,X_f ← E)
        →   % The candidate is consistent
            (var(X_f)
            →   suspend
            ;   true)
        ;   set_dependency(X_a,X_f,E),    % removes the inconsistent value
            lazy_constraint(X_a,X_f)).    % looks for another consistent value
```

Figure 14.3 A MFC CLP Implementation - Example of binary constraint

This implementation of MFC has a slightly different behavior w.r.t. the algorithm described in [23]; in fact, it does not record information on successful constraint checks. Only unsuccessful constraint checks produce a domain modification, so we think that they are more appropriate in a CLP framework. However, information about successful constraint checks can be inserted and checked in the `is_consistent` predicate.

Moreover, in order to minimize the number of constraint checks, the MFC algorithm needs to activate the constraints in the instantiation order. E.g., suppose that the current variable X_a is assigned a value and removes the candidate for the future variable X_f. The next value (say value v) must be checked for consistency w.r.t. all the past connected variables: suppose that it is inconsistent with the constraints $c(X_1, X_f)$ and $c(X_4, X_f)$. If the constraint $c(X_1, X_f)$ is activated first, the element will not be re-inserted until X_1 is uninstantiated; if $c(X_4, X_f)$ is activated first, then the culprit of the elimination will be X_4, so value v will be re-inserted when X_4 is uninstantiated. However, the algorithm correctness is independent on the activation order, so we consider this issue as a matter of the heuristics used to activate the constraints [114].

Finally, note that lazy constraints can be mixed with eager constraints without affecting correctness, but only efficiency. So, we can exploit laziness for expensive constraints (constraints in which each constraint checking is hard) and eagerness for efficient constraints.

It's worth noting that, thanks to this predicate, we do not have to use extralogical predicates to implement this algorithm. In plain Prolog, we would have need of extralogical predicates, because we need to record information about deleted values after backtracking,

but no logic construct of Prolog survives backtracking. In the next section, we show a simple meta-interpreter providing the depicted functionality.

14.3 A simple Meta-interpreter

To keep information after a backtracking, the only methods provided by Prolog are the `assert/1` and the `retract/1` predicates. For this reason, the `set_dependency(A,Var,Val)` predicate asserts a fact `dep(A,Var,Val)` stating that the removal of a the element `Val` from the domain of variable `Var` depends on the assignment on the variable `A`.

In order to have a backtracking rule exploiting the information recorded in the `dep/3` structures, we developed a simple meta-interpreter (Figure 14.4), based on the idea that after backtracking, we redo all the computation that had not to be undone. In other words, after a backtracking, Prolog undoes an instantiation and all the chronologically successive computation; since we want to undo only the computation that *depends* on the instantiation, the rest has to be redone. This operation is necessary because in a Prolog meta-interpreter we do not have direct access to the operations performed during the unlabelled phase. For this reason, the meta-interpreter generates two events, called forward and backward event. The forward event is generated just before performing a choice. Since in Prolog choices are made when we select the head of a clause, the meta-interpreter triggers this event before selecting a clause for unification. The forward event redoes all the computation which has been unnecessarily undone. The backward event is activated if unification leads to failure. In the backward event we retract all the `dep/3` facts depending on the variables that are no more instantiated.

Note that the space complexity of the additional data is $O(nd)$ (where n is the number of CSP variables and d is the maximum domain size) as in the most efficient MFC implementation [3].

If we consider the number of constraint checks (Figure 14.5.a), we can see that MFC with chronological backtracking performs much worse than our version of MFC. Comparing the classical MAC algorithm usually employed in CLP systems with MFC (Figure 14.5.b), we can see that the efficiency gain is even better. In these figures, the area painted in black shows the set of problems that are solved best with chronological backtracking. As the graph goes higher, our backtracking rule outperforms the chronological backtracking.

```
% solve(Goal List,List of the CSP variables)
solve([],_) :- !.
solve([A|B],L) :- !, solve(A,L), solve(B,L).
solve(H,L) :- functor(H,Func,Arity), functor(Templ,Func,Arity),
              clause(Templ,Body),        % select a clause for unification
              (fwd_event(L) ; bck_event(L), fail),
              H=Templ,                   % perform unification with the selected clause
              solve(Body,L).

fwd_event(L) :- ( redo_flag              % If we are in a redo phase,
                  →  redo_unback(L),
                     reset redo_flag
                  ;  true).
bck_event(L) :- forget_dep(L),           % retract all the dep/3 facts which depend
                set redo_flag.           % on the instantiation of the variables
                     we are uninstatiating

% delete from the domains of the variables in VarList
% all the values declared in dep/3 facts.
redo_unback(VarList) :-
      findall([Var,Val],dep(_,Var,Val),DeleteList),
      redo_deletions(DeleteList,VarList).

redo_deletions([],_).
redo_deletions([[Ref,Val]|T],VarList) :-
      match_variable(Ref,VarList),
% get the variable given its reference and the list of CSP variables
      Var ≠ Val, redo_deletions(T,VarList).

forget_dep([]).
forget_dep([var(_,Val)|T]) :-
      nonvar(Val), forget_dep(T).
forget_dep([var(N,V)|T]) :-
      var(V), retrieve_ref(V,Ref), retract_all(dep(Ref,_,_)), forget_dep(T).

set_dependency(A,Var,Val) :- retrieve_ref(A,RefA), retrieve_ref(Var,RefV),
      assert(dep(RefA,RefV,Val)).
```

Figure 14.4 Metainterpreter for the implementation of the `set_dependency` predicate

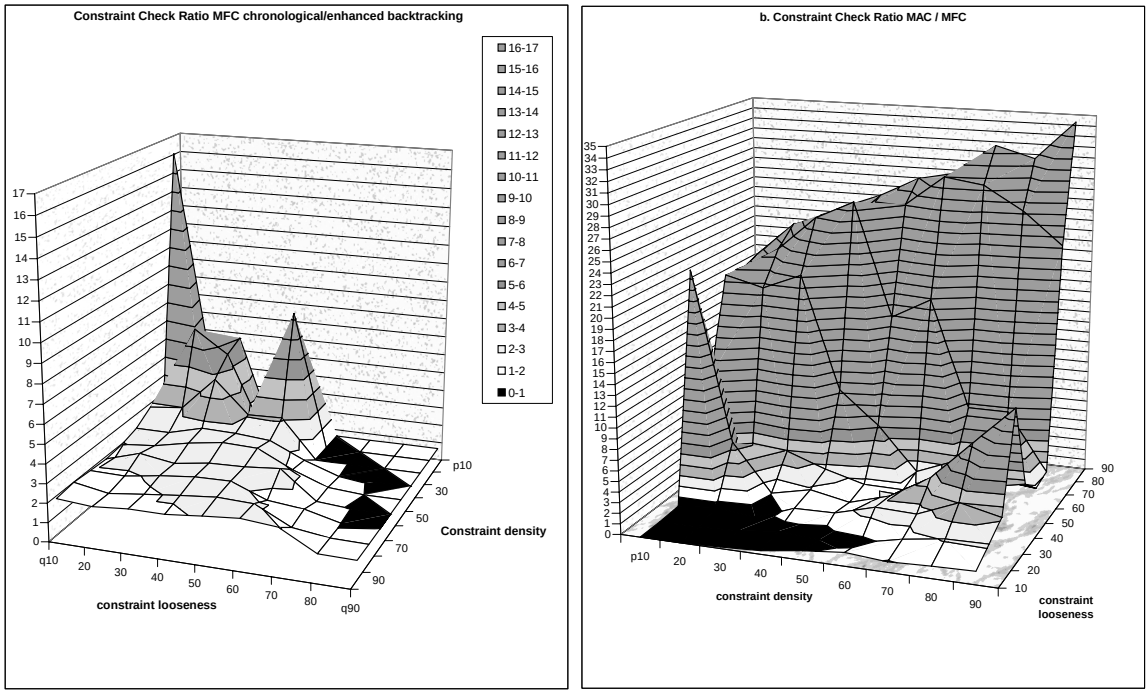

Figure 14.5 Constraint check ratios

Timing results (Figure 14.6)[1] show a smaller improvement, because we wanted mainly to obtain a simple implementation, and it is not optimized, so the introduced overhead is significant. Since this overhead does not affect the number of constraint checks, it cannot be seen in the graphs of Figure 14.5.

[1]For the sake of fairness, also the algorithms exploiting chronological backtracking were meta-interpreted.

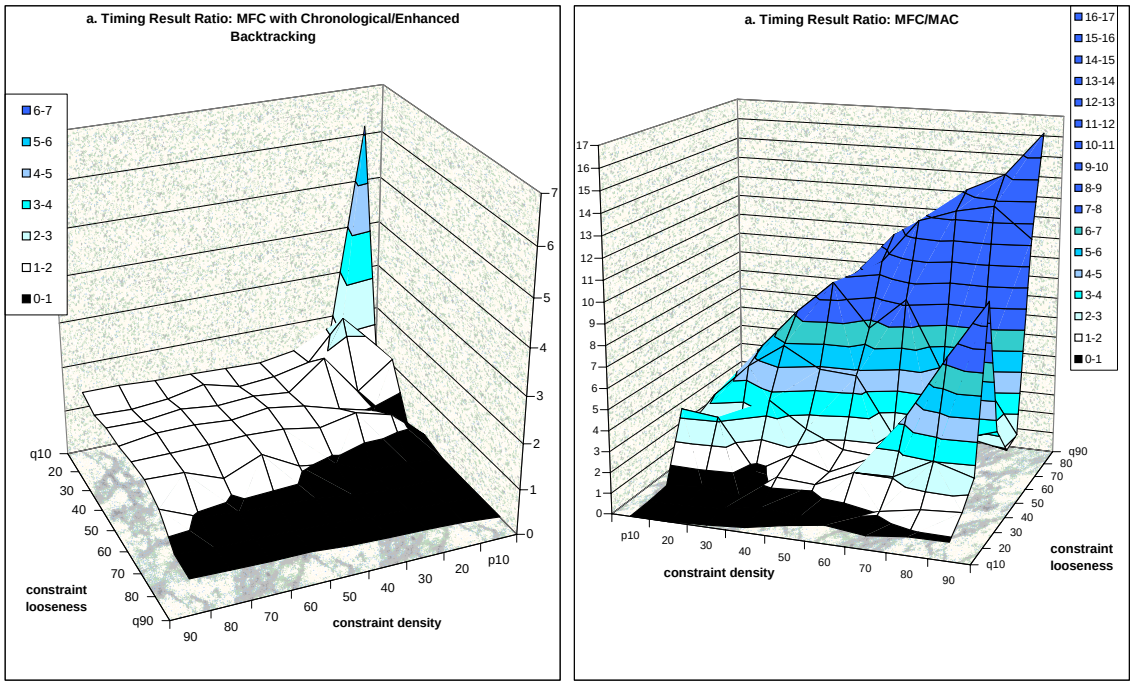

Figure 14.6 Timing Results

15.0 Discussion

In this section, we discuss some points concerning: (*i*) when to use ICSP instead of CSP; (*ii*) when to use either an eager or a lazy or an intermediate form of acquisition; (*iii*) which search strategies are more suitable within this framework.

15.1 CSP vs. ICSP

In general, using the ICSP instead of standard CSP framework is convenient for certain classes of applications. For instance, it is convenient when we look for one feasible solution, while for applications where all solutions are required, the ICSP can be more suitable only if constraints are very tight. Constraint tightness is an important parameter that should be taken into account. For tight constraints the ICSP approach outperforms classical CSPs since, during search, we can work with domains smaller than those managed by pure CSP approaches. For the same reason, other applications which benefit from the proposed solution are those where domains contain a huge number of values. Clearly, the acquisition of domain values is an important parameter that enables us to decide whether or not to use the ICSP framework. When acquisition is computationally expensive, or in general more expensive than the constraint solving part, the interactive approach represents the technique of choice. Furthermore, when domain data are not known in advance, the ICSP is a promising way of addressing the problem.

Default domains. In most constraint programming systems [2, 24, 91, 93], solutions have been proposed to let the variables range on some *default domain* (usually $[-Max, \ldots, +Max]$ for integer domains) when domains are unknown. In many cases, default domains lead to very inefficient or very unnatural solutions. Consider, for example, a visual search problem: the variables range over the set of surfaces taken from an image by a segmentation module. Each surface is described by a set of parameters; for example, for each surface belonging to a range image, we could have the direction of the normal, the centroid coordinates, the list of vertices. Thus, the default domains are unmanageably large, as they must contain each possible surface an image can contain. Since the centroid coordinates range on $\mathcal{R}^2$, the normal ranges on $\mathcal{R}^3$ and the list of vertices is a collection of points in the image ($\mathcal{R}^2$), the default domain should be $\mathcal{R}^2 \times \mathcal{R}^3 \times \mathcal{P}_{\mathcal{R}^2}$. In this case, there would be a constraint stating that each variable should represent a surface in the actual

image; this constraint could be checked by the segmentation unit, in a concurrent constraint computation [100].

15.2 Parameter tuning

Having selected the ICSP as the technique for solving the problem, we have to tune the algorithm parameters. Depending on the application, a lazy approach may be more convenient than an eager one or vice versa. An eager acquisition is convenient in applications where: (*i*) the communication between the producer and the consumer of domain values has a high overhead; (*ii*) acquiring one or all consistent domain values has almost the same complexity; this is true in applications where constraints define a sort of locality criterion which can be exploited in the acquisition. For instance, in the visual search application, the constraint *touch* allows the producer of domain values, i.e., the image processing module, to focus its attention on a subset of the image near a given surface. (*iii*) Interactive constraints are tight, so the number of retrieved values is significantly smaller than the total number of values of the complete domain; (*iv*) the problem is not completely connected. If each variable is linked to each other, as soon as the first value is instantiated, the IFC retrieves all consistent values for each other variable, thus leading to a standard Forward Checking algorithm.

On the other hand, a lazy acquisition is preferable when the acquisition of constrained data is expensive and, in particular, is proportional to the number of retrieved values, and when the problem is completely connected.

The intermediate uncoupled version, instead, can be used when the producer of constrained data is a black box system, a commercial tool which has predefined ways of exploring a given environment or predefined ways of being guided. Thus, in this case the constraint solver has to check retrieved data. If the acquisition module can be guided to acquire data, a translation between interactive constraints and guidelines must be provided.

15.3 Heuristics

Other parameters of the algorithm are the heuristics for variable and value selection. As concerns variable selection, a widely used CSP heuristics, called *first fail principle* [55], selects first the variable with the smallest domain. When coping with partially or fully

unknown domains, we cannot know in advance how many values will be contained in the domain. Thus, we partially disable variable selection heuristics which depend on domain size. A general criterion which can be followed in the interactive framework tends to minimize knowledge acquisitions. Thus, we first select variables with a completely known domain, then partially known variables, and finally, completely unknown variables.

As concerns the domain value choice heuristics, values in the known domain part are preferable, as they do not require an unguided acquisition.

In some cases, when partial knowledge is available, it can be more convenient to generate a classical CSP working on available domain data. If the computation fails new information can be retrieved and a new CSP computation can be started from scratch. Thus, we can solve each time a new CSP with more and more information. In our framework, this behavior is allowed and can be further improved by simply using a particular branching strategy. For each selected variable X ranging on a partially known domain $X :: \{v1, v2, v3|U_X\}$, we propose a branching rule imposing $X :: \{v1, v2, v3\}$ as a first choice (i.e., *closing* the domain), and $X :: U_X$ as a second. Intuitively, this strategy implies that we first try to solve a standard CSP (all variable domains are *closed*) by using available domain values and if the constraint solving part fails, we retrieve more data and continue the computation. In this setting, the intermediate approaches to information extraction retrieving each time M values are more suitable. In fact, we can retrieve a bunch of values, try to find a solution, and if it is not the case, try to add another bunch to a given domain and so on.

16.0 Measuring Interactive Algorithms Performance

The efficiency of an ICSP solving algorithm can be estimated by considering a combination of different computation/communication times [46]. Since the whole system is composed of two basic communicating subsystems, we need to consider the computation performed by the ICSP solver, the one performed by the extraction agent and the communication cost. Then, the overall computation time can be calculated if we know the interaction protocol. In other words, if we have a measure, m_{ICSP}, of the ICSP solver computation time, a measure, m_{acq}, of the acquisition agent computation time and a measure, m_{com}, of the communication cost, then the overall computation time can be calculated by $t = f(m_{ICSP}, m_{acq}, m_{com})$, where f depends on the interaction protocol. In the next sections, we show how to compute the three given parameters, plus the combination function, for a given problem.

16.1 Esteem of the ICSP solver computation time

The ICSP solver's performs a computation with many similarities to a standard CSP solver. In fact, the ICSP solver can be implemented on top of a CSP solver [19] or on a stand-alone language. Thus, the same measures used in the CSP field can be employed to obtain the ICSP solver computation time. As in many works [95, 108, 23] we employed the number of constraint checks as a measure of the ICSP solver computation time:

$$m_{ICSP} = c_c^i n_{chk}^{ICSP}$$

where c_c^i is the (average) time employed to perform a constraint check and n_{chk}^{ICSP} is the number of constraint checks performed by the ICSP solver.

16.2 Esteem of the extraction agent computation time

In our framework, the value extractor extracts values and also performs a constraint solving computation; in fact, it retrieves consistent values. This constraint checking can be requested to the low-level system instead of the high-level system because of different reasons. It can be more *convenient* because a particular type of constraint is checked more efficiently by the low-level system, or because the low-level system *knows* more about the

problem than the high-level system (i.e., the high-level system does not know how to check a particular constraint), or because the high-level system does not know the (domain) elements. For this reasons, the number of (low-level) constraint checks is also a good indicator of the amount of computation performed by the low-level system.

In other systems, the time employed by the low-level system is represented by the number of extracted elements. In fact, in some cases, the operation of constraint checking is very simple from the low-level system viewpoint; instead, extracting a new domain value is very difficult.

As a conclusion, the computational effort of the extraction agent can be considered as a linear combination of the number of extracted elements, n_{extr} and the number of constraint checks performed by the low-level system, n_{chk}^{acq}. In other words, there will exist two coefficients c_{ll}^{acq} and c_{extr}^{acq} such that:

$$m_{acq} = c_{ll}^{acq} n_{chk}^{acq} + c_{extr}^{acq} n_{extr}$$

Considering the problem in higher detail, we need to take into account how the search space is explored by the extraction agent. If one value is searched at a time, then we need to consider the eventual search for an element that does not exist. In this case, if the search space contains n elements consistent with the imposed interactive constraints, the computational effort is roughly proportional to $n + 1$, because, after the last element is acquired, another one is looked for (and not found).

On the other hand, if the interactive constraints can effectively be employed to index the search space and *a priori* eliminate all the inconsistent elements, then there will not be a computational effort to search for the "after-last" element.

In the following, we will consider the extra-check, because we decided to perform a worst-case analysis, that is very significant in time-consuming applications, providing an upper bound to the computation time.

16.3 Esteem of the communication cost

Usually communication time between subsystems depends on the size of the transmitted information plus a (roughly fixed) overhead time for each message. In our case, there are two kinds of messages: requests and replies.

The request message contains the interactive constraints employed for acquisition. In this work, we made a simplifying assumption: elements can be requested by means of one constraint at a time. In other words, we considered a simple low-level system that is not capable of acquiring values consistent with more constraints. Of course, the ICSP model benefits from systems that are able to employ all the imposed constraints, obtaining thus a high level of consistency [33]; again, we preferred a worst-case analysis.

With this assumption, the size of the request message is constant, so the cost of the request messages is proportional to the number of messages: $m_{requests} \propto n_{req}$.

The size of a reply message is not fixed, as the number of retrieved values varies. The cost of a reply message is the sum of a constant part (due to the header of the message) plus a quantity proportional to the number of elements carried by the message. Since for each request there is a reply, the total overhead due to the transmission of the constant parts is proportional to the number of request messages. The total transmission cost of the variable parts depends on the total number of transmitted values; so there are two coefficients c_{fix}^{com} and c_{elm}^{com} (the latter representing the average transmission cost of a domain element) such that the global communication cost is given by:

$$m_{com} = c_{fix}^{com} n_{req} + c_{el}^{com} n_{tr}$$

where n_{req} is the number of requests to the acquisition agents and n_{tr} is the number of transmitted elements.

Calculating the number of transmitted elements depends on the type of the application. If extraction is an expensive task, it must not be done twice (e.g., when exploring different branches in backtracking); retrieved elements have to be memorized. If all the constraint checking activity can be carried out by the ICSP solver, then retrieved values will be probably kept in memory by the ICSP solver itself and the acquisition agent will have to provide only new values. In this case, the number of transmitted elements will be equal to the number of extracted elements: $n_{tr} = n_{extr}$.

On the other hand, if the ICSP solver is not able to perform some kinds of constraint checks (as efficiently as the acquisition agent is), then it will probably let the acquisition agent do this task. In other words, it will not memorize the acquired domain elements, and will ask the acquisition agent for doing so. In this case, we can say that each time an element is transmitted, it had been checked for consistency by the extracting subsystem. This means that $n_{tr} \leq n_{chk}^{acq}$.

To sum up, there are two values α, β such that $n_{tr} \leq \alpha n_{extr} + \beta n_{chk}^{acq}$; so:

$$m_{com} \leq c_{fix}^{com} n_{req} + c_{el}^{com}(\alpha n_{extr} + \beta n_{chk}^{acq}) = c_{fix}^{com} n_{req} + c_{extr}^{com} n_{extr} + c_{ll}^{com} n_{chk}^{acq}$$

16.4 The Combination Function

As we noticed before, the total cost of the algorithm is a function of the three described values: m_{ICSP}, m_{acq}, m_{com}. If the subsystems are not able to work in parallel, but the computation is alternate, then the total cost is given by the sum of the three costs. In fact, the ICSP solver will request some consistent values, the acquisition agent will retrieve the values and will reply. Only when the ICSP solver receives the domain elements it can proceed with its computation. So,

$$t_{tot} = m_{ICSP} + m_{acq} + m_{com} =$$

$$\leq c_c^i n_{chk}^{ICSP} + c_{ll}^{acq} n_{chk}^{acq} + c_{extr}^{acq} n_{extr} + c_{fix}^{com} n_{req} + c_{extr}^{com} n_{extr} + c_{ll}^{com} n_{chk}^{acq} =$$

$$c_c^i n_{chk}^{ICSP} + c_{ll}^{tot} n_{chk}^{acq} + c_{extr}^{tot} n_{extr} + c_{fix}^{com} n_{req}$$

At the opposite, if the subsystems are able to work in parallel such that one system has never to wait for the other one, then the communication cost is masked, because the communication is performed while the two subsystems are working. Moreover, the total computation time will be given by the maximum of the computation times:

$$t_{tot} = Max(m_{ICSP}, m_{acq}) = Max(c_c^i n_{chk}^{ICSP}, c_{ll}^{acq} n_{chk}^{acq} + c_{extr}^{acq} n_{extr})$$

17.0 Implementation and Experimental Results

We implemented the ICSP framework on top of the FD library of the ECL^{i}PS^e system [2]. First we redefined the concept of domain in order to accept value acquisition. A new event triggering the propagation has been added to consider value acquisition. When this event is raised, the constraints that have to test the acquired values are awakened. Finally, we added facilities to define new constraints, to add customizable interactive labelling predicates and to give information about the domains to allow the user defining his own heuristics [19].

Further details on ICSP plus the implementation can be found at the web site:
`http://www-lia.deis.unibo.it/Research/Areas/icsp/icsp.html`.

We tested the described algorithms on a series of randomly generated problems, as proposed in [95], and considered the average of some significant indexes. A CSP can be generated by considering four parameters: the number n of variables, the size d of each domain, the probability p that a constraint exists on a given couple of variables and the conditional probability q that a couple of assignments are consistent given that the variables are linked by a constraint. We generated the problems considering $n = 10$ variables and $d = 10$ as domain size and varying the constraint density p and the constraint looseness q from 10% up to 90%.

As we described earlier, an ICSP can be solved either by passing all the interactive constraint to an acquisition module, or by performing acquisition only with few of them. In order to perform a worst-case analysis, in the experimental results, we considered acquisition driven by only *one* constraint at a time.

The main purpose of an ICSP solving algorithm is the minimization of the need for domain elements without having to restart constraint propagation from scratch. Domain elements are hard to acquire, because they come from the user, or from systems that synthesize symbolic information from signals and sensors. In our simulations, we considered thus the number of extracted elements as the most important efficiency index. In figure 17.1 the percentage of extracted elements is shown for IFC and in figure 17.2 for IMFC. We can see that IFC needs, in order either to find a solution or to prove infeasibility, about 50% the domain elements required by a non interactive algorithm. IMFC needs nearly one third, and in more than 30% of the generated problems, it needed less than 20%. In particular, IMFC proves more effective than IFC when constraints are loose (high values for q), because

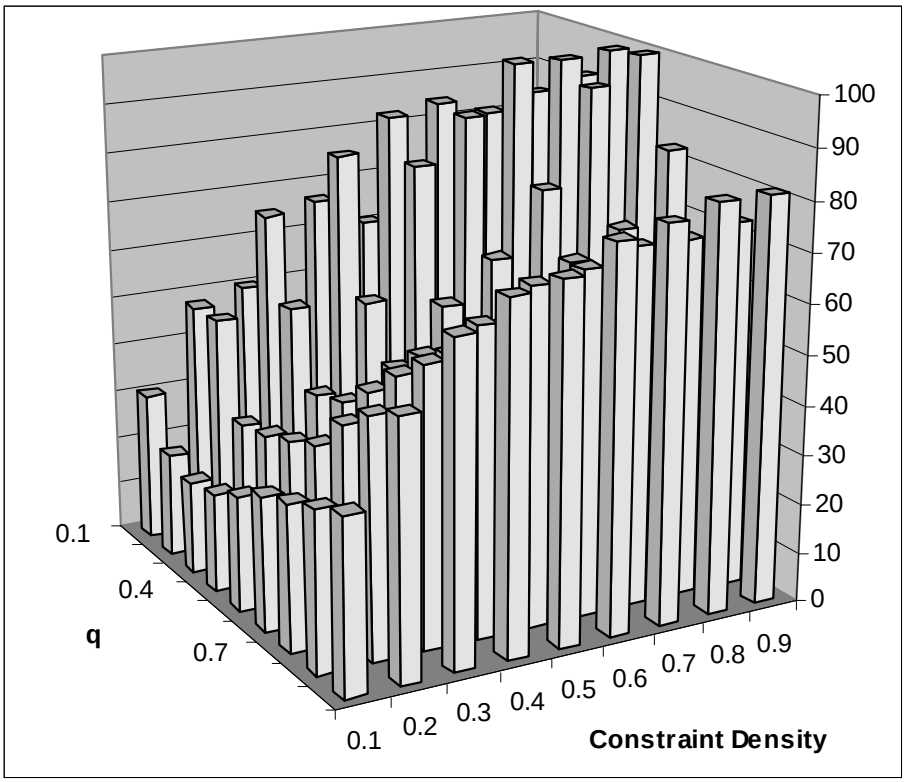

Figure 17.1 Interactive Forward Checking - Percentage of extracted elements

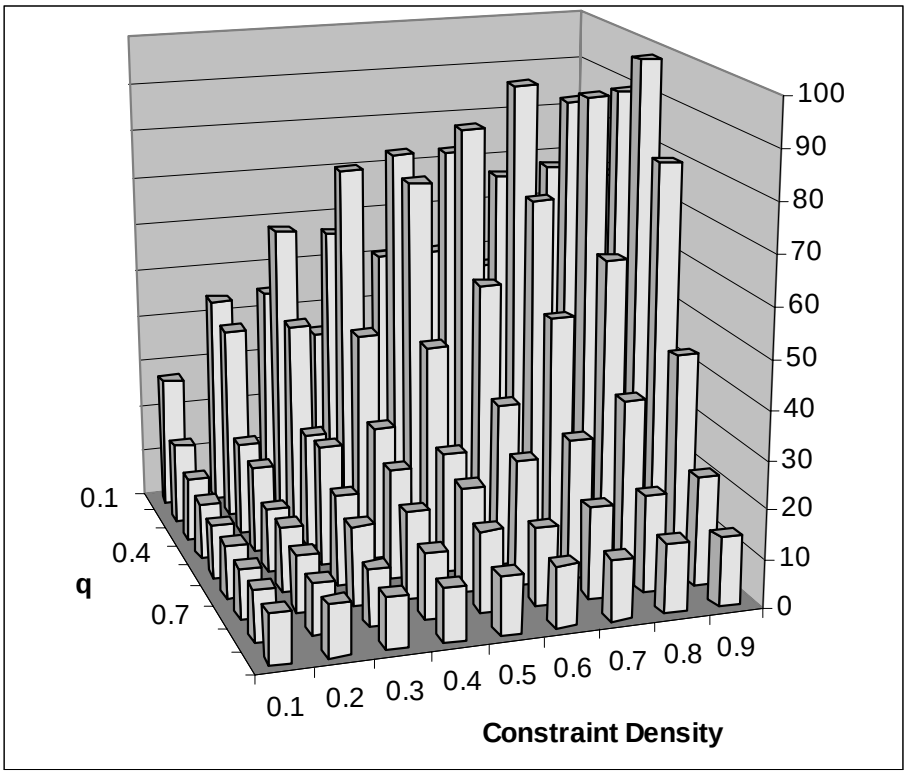

Figure 17.2 Interactive Minimal Forward Checking - Percentage of extracted elements

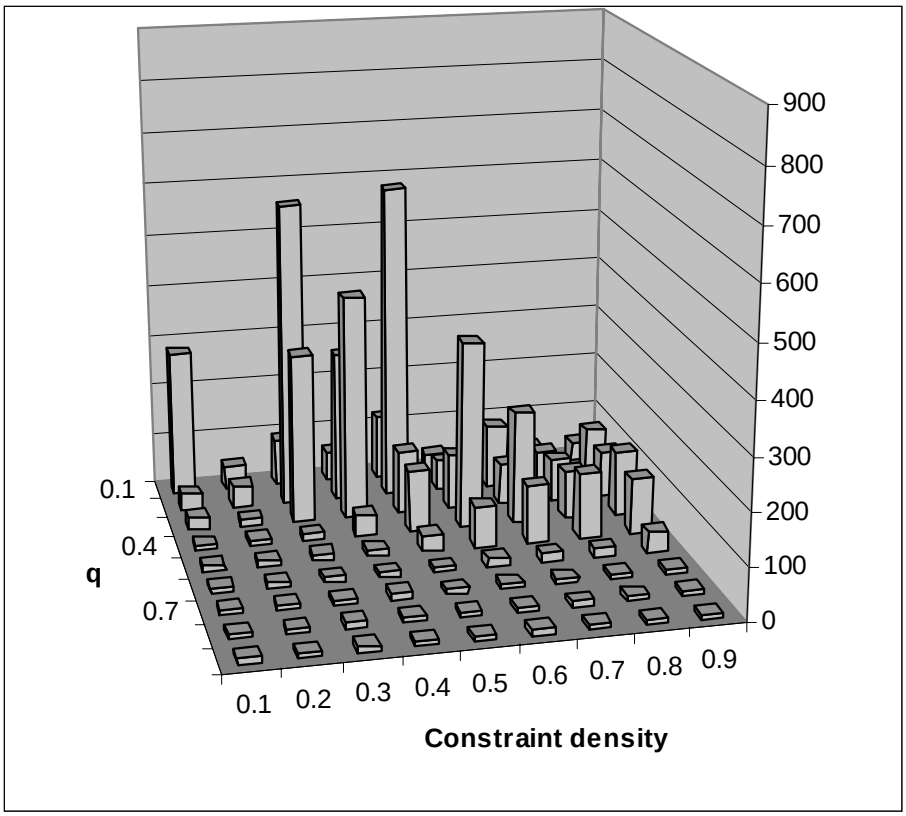

Figure 17.3 Interactive Forward Checking - Mean number of interactions n_{req}

it limits acquisition to just the first consistent element, while IFC requires all the consistent elements. On the other hand, acquiring only one value at a time may mean performing many acquisition steps.

If communication between the subsystems is costly, (e.g., if the subsystems are far, and have to communicate by means of a wide area network), the transmission time should be taken into account. Usually communication time depends on the size of information (that can be considered proportional with the number of domain elements in each message) plus a fixed time for each message.

In particular, request messages have all the same size, so their total cost t_{req} is proportional to the number of messages n_{req}: $t_{req} \propto n_{req}$. For each request there is a reply; reply messages carry the extracted elements, so the total communication cost is $t_{com} = c_{fix}^{com} n_{req} + c_{extr}^{com} n_{extr}$ (for some coefficients c_{fix}^{com} and c_{extr}^{com}). For this reason, we counted the number of *interactions* (i.e., request messages n_{req}) needed to solve the given

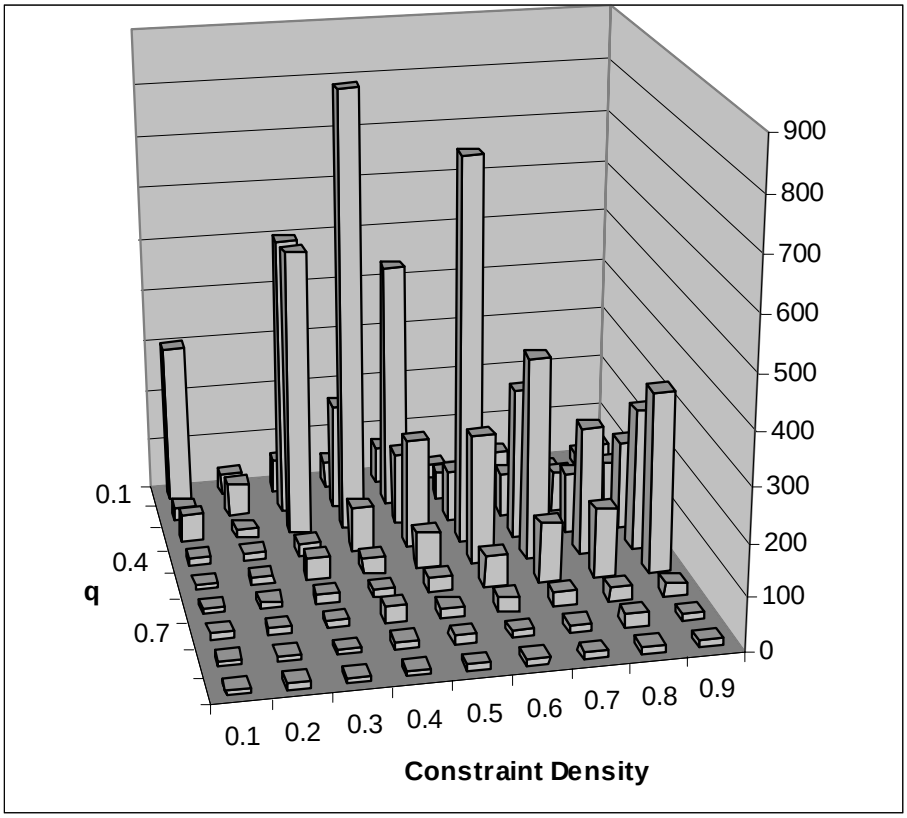

Figure 17.4 Interactive Minimal Forward Checking - Mean number of interactions n_{req}

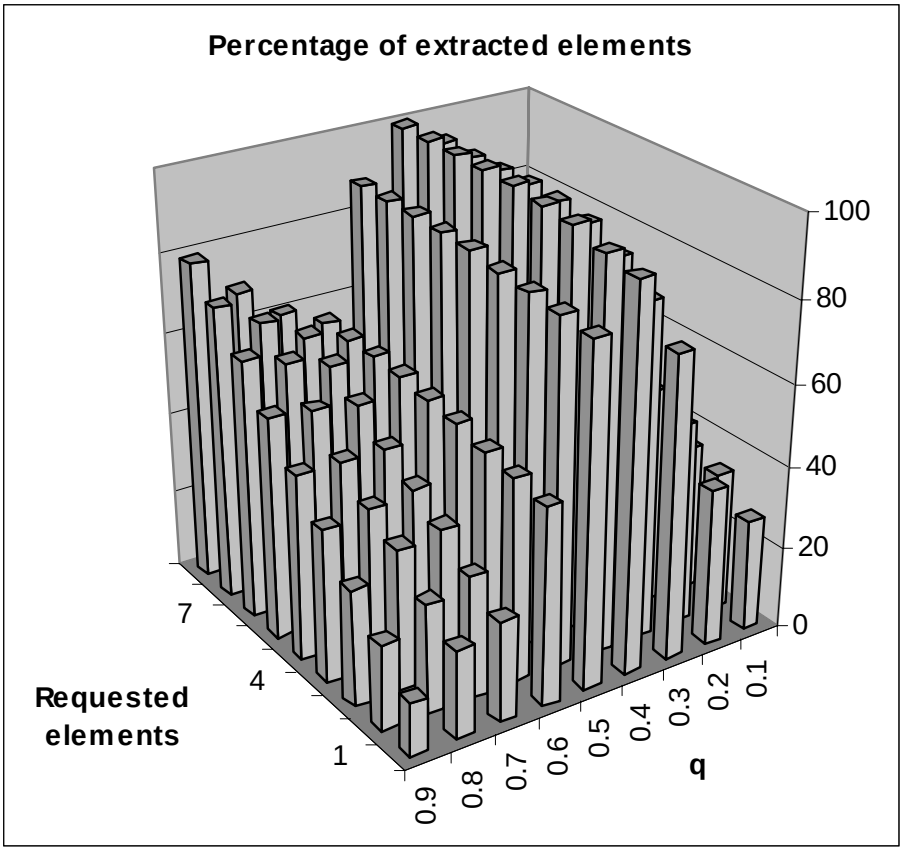

Figure 17.5 Extractions varying the number of extracted elements

problem, and reported it in Figure 17.3 for IFC and in Figure 17.4 for IMFC. As we expected, the number of interactions needed by the IMFC algorithm is usually higher. This makes IFC more suitable when the communication cost is high.

As explained in previous paragraphs, previously acquired domain values are kept in memory, in order to avoid acquisition of already extracted values. This memory can be used as a "prefetching" buffer, where the acquisition subsystem puts extracted values not yet requested.

By considering this, the IMFC algorithm can be improved. In fact, acquisition of just one domain value can be unwise if more consistent values are already available, because we risk to unnecessarily perform many acquisition steps. For this reason, we implemented and tested an interactive algorithm that retrieves a number of domain values given as parameter; the results are shown in Figures 17.5 and 17.6. In these graphs, we considered a constraint density $p = 70\%$ and varied the constraint looseness q and the number of requested elements.

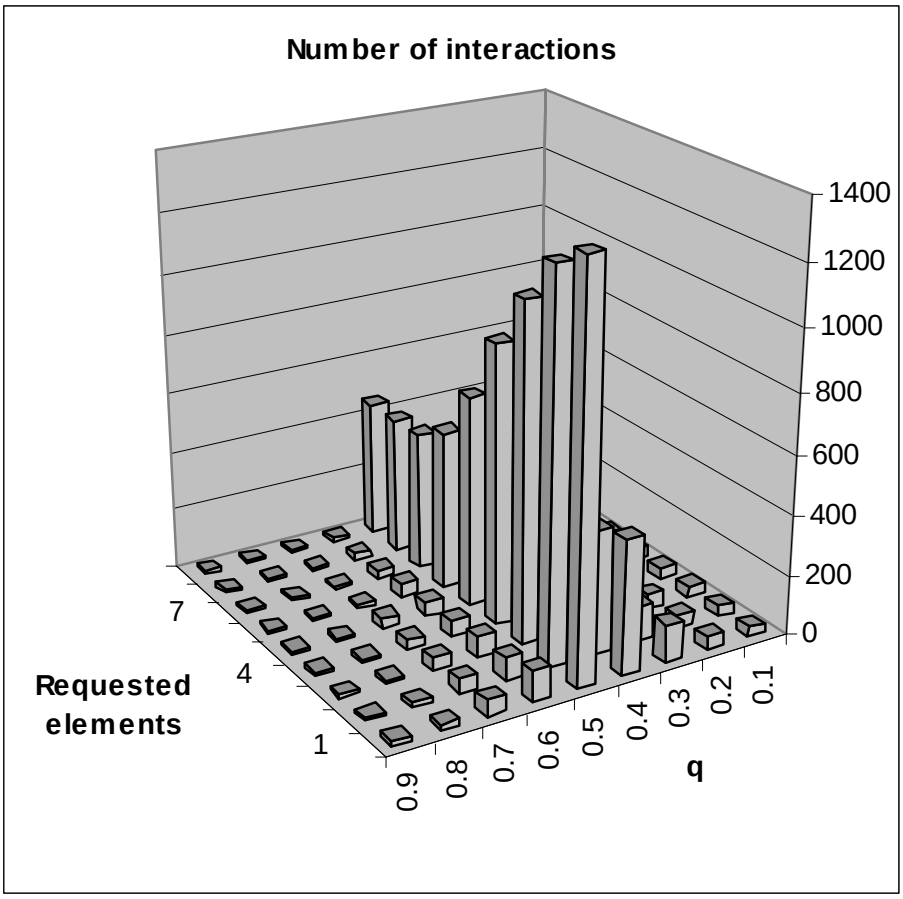

Figure 17.6 Interactions varying the number of extracted elements

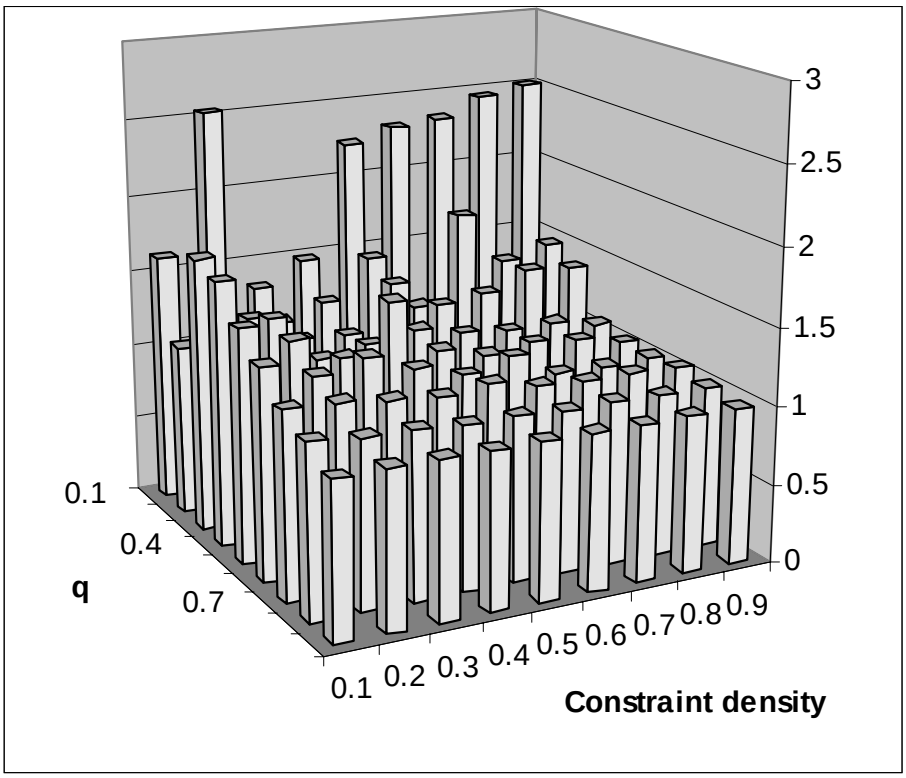

Figure 17.7 Constraint check ratio FC/IFC

As the number of requested elements grows, the behavior of the algorithm resembles the IFC algorithm: the number of extracted elements grows (Fig. 17.5), while the number of interactions (Fig. 17.6) needed to get to a solution decreases.

Of course, all these considerations should consider the computation time of both the extraction module and the ICSP solver. Since they both perform constraint checks, we counted the number of constraint checks performed by both systems. If the number of constraints checked by an ICSP based system was much worse than a CSP based solution, then the ICSP would be outperformed even when extracting few elements. We studied the number of constraint checks and found that an ICSP solution takes about the same number of constraint checks as a CSP solution. In Figure 17.7 the ratio of constraint checks is shown. Here the same problem was solved by a Forward Checking algorithm and by the Interactive Forward Checking algorithm; the higher the bars, the better the interactive approach performs. In figure 17.8 the interactive and standard versions of Minimal Forward Checking are compared, considering the number of constraint checks. The number

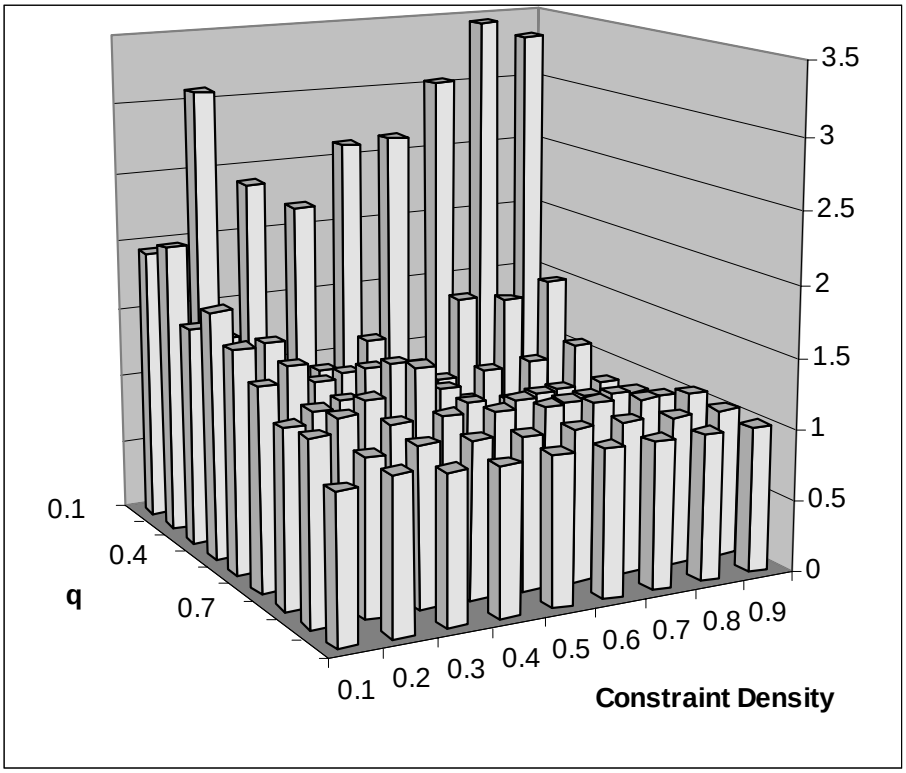

Figure 17.8 Constraint check ratio MFC/IMFC

of constraint checks is usually lower in the interactive approach because many constraint checks are avoided due to the focalization. In fact, in many cases, as in a vision system [19], when searching for a new element most of the values are a priori discarded. For example, if we know in which part of the image a visual feature is located, we do not need to check all the possible features, but only the ones in the selected location. For this reason, the acquisition system does not check many domain values (all the inconsistent ones) and we did not count those checks in the simulations. All the checks were counted by using a memorization of successful/unsuccessful checks [23] [3]. However, the main issue of this work was not minimizing the number of constraint checks, but the number of domain value acquisitions, because each acquisition is expensive (can involve information extraction) while a constraint check is simply a table lookup. Some overhead is introduced, so, if acquisition is not expensive, a total extraction followed by a classic CSP resolution can yield a better performance.

As a conclusion, the total execution time, if all the operations are executed in an alternate computation, can be estimated with the sum of the computation times of the ICSP solver, the acquisition module and the communication cost: $t_{tot} = t_{ICSP} + t_{acq} + t_{com} = c_c^i n_{chk}^{ICSP} + c_{ll}^{acq} n_{chk}^{acq} + c_{extr}^{tot} n_{extr} + c_{fix}^{com} n_{req}$ (where n_{chk}^{ICSP} is the number of constraint checks done by the ICSP solver, n_{chk}^{acq} is the number of constraint checks performed by the acquisition module and the c are coefficients). If we consider equal computation time for constraints checked by the ICSP solver and the acquisition agent, (i.e., $c_c^i = c_{ll}^{acq}$), the total computation time can be rewritten as: $t_{tot} = c_{chk} n_{chk} + c_{extr}^{tot} n_{extr} + c_{fix}^{com} n_{req}$. In Figure 17.9 we considered the average of the total computation time of the various algorithms varying the relative costs of extraction, interaction and constraint checking and we showed the best algorithm for each case. Considering the presented results, we can see that the interactive algorithms are usually very promising, unless the communication cost is prevailing. As the communication cost decreases, Interactive Forward Checking becomes more attractive and, if the communication cost is very low, the IMFC algorithm is more suitable (see Figure 17.9). On the other hand, if the extraction cost is very low, then standard algorithms can be more effective, since the ICSP framework needs to redefine the constraint solver engine. As the extraction time grows, the IFC algorithm performs better and, finally, if the extraction time is very high, the IMFC algorithm allows a minimization of the number of extracted elements.

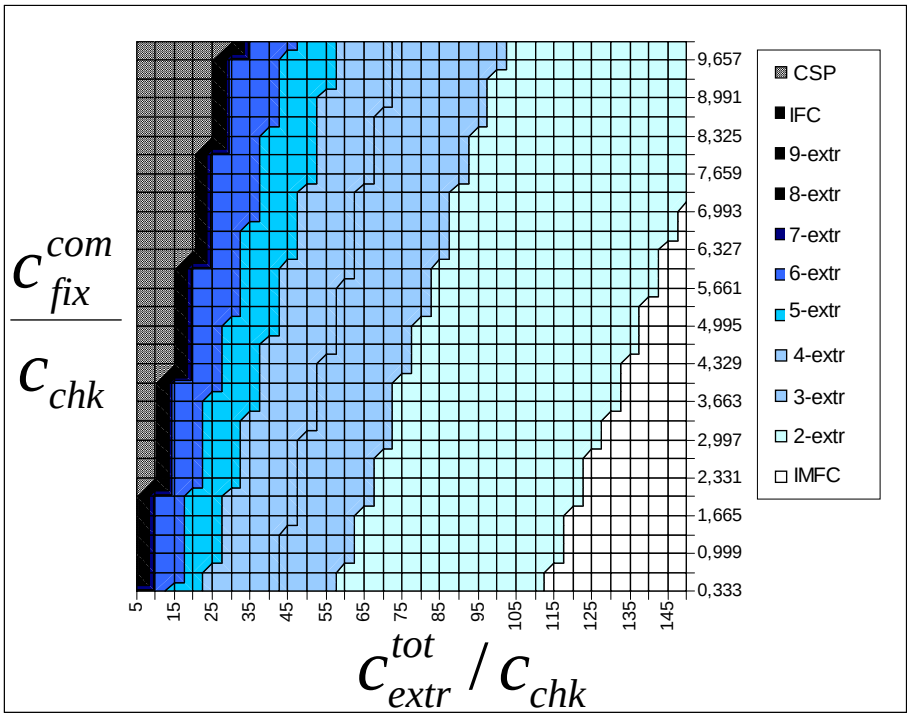

Figure 17.9 Most efficient algorithm varying the relative costs of extraction, interaction and checking

18.0 Applications

In this section, we discuss some applications which can benefit from the proposed framework. In general, the ICSP framework can be used for modelling and solving those applications where the acquisition of domain values is expensive.

18.1 Visual search

One of the most successful applications of the ICSP is in the field of 3D visual object recognition [19]. Variables represent object parts (say surfaces) and constraints represent geometric and topological relations among them. Solving the 3D object recognition problem as a standard CSP involves extracting all surfaces from the image which are put in variable domains. An object is recognized in a scene if we find an assignment of values to variables satisfying all the constraints.

However, surface extraction is a time consuming step even working on range images. Real images are more complex, as we have reflection, diffraction and shadows. Thus, useless acquisitions should be avoided. The adoption of ICSP leads to a significant performance improvement due both to the constraint solving part which has to manage smaller domains and, more importantly, to the visual system which focuses only on significant image parts. In fact, the guided interaction prevents the low-level system from acquiring useless surfaces. Several tests have been performed on a specific data-base of range images and are reported in Table 18.1. For each image, the time employed by an ICSP solver exploiting interactive constraints for acquisition and the corresponding CSP solver are compared. Both the solvers stop when the desired object is found. In most cases, a good speedup is provided due to the fact that the CSP solver has to acquire all the features before constraint propagation, while the ICSP interactively acquires only the needed features. Only in one image (called "BLOCK_8") the ICSP is slower than the corresponding CSP solver, because the requested object was not present in the image. In this case, all the image features had to be acquired and the ICSP solver has a small overhead.

Moreover, interactive constraints could be used to communicate different types of information to the segmentation system. For example, very inclined surfaces are difficult to extract, because few pixels are available. For the segmentation of inclined surfaces, different

Image	ICSP	CSP	Speedup
BLOCK_1 (320x320)	136.60	279.66	2.04
BLOCK_2 (320x320)	129.80	276.07	2.12
BLOCK_3 (320x320)	125.01	256.51	2.05
BLOCK_4 (320x320)	39.93	263.50	6.59
BLOCK_5 (320x320)	156.50	309.90	1.98
BLOCK_6 (320x320)	34.89	301.43	8.63
BLOCK_7 (400x400)	178.77	442.51	2.47
BLOCK_8 (400x400)	549.10	518.59	0.94
BLOCK_9 (400x400)	215.85	555.76	2.57

Table 18.1 Experimental results of an ICSP-based image recognition system

parameters or different algorithms could be employed, to achieve a more reliable recognition. So, if the constraint solver needs to acquire a surface that is perpendicular to S_1, and S_1 is not very inclined, the constraint solver could infer that the acquired surfaces will be very inclined; the interactive *perpendicular* constraint could be sent to the acquisition system that could use an algorithm suitable for inclined surfaces.

18.2 Planning

Another application where the ICSP has been successfully applied concerns planning. A Partial Order Planner has been extended [118] performing least commitment on variable binding for coping with incomplete knowledge. The planner has been described in [5] and applied in the field of distributed computer system management. In this domain, data on system resources and services are huge, heterogeneous, hierarchically organized, hard to be managed and efficiently stored. Thus, in the description of action schemata, variables representing resources are associated with an unknown domain of values which are retrieved by the planner during plan construction. The planner retrieves only those values which are really needed for the plan construction. The acquisition part of ICSP has been implemented via operating system scripts aimed at retrieving only values consistent with interactive constraints or via queries to the system administrator with a user friendly interface.

In general, those applications managing a large amount of data can be faced with the ICSP framework. Consider, for instance, a problem where data is stored in a database. Classical CSPs should collect the whole content of the database, store data in variable domains, and perform the constraint solving process. Each constraint c defines a view on

the database containing only the set of tuples T satisfying c. By using the ICSP approach, each interactive constraint can be mapped into a query to the database management system selecting only the set T. Query languages provide a powerful tool for performing conjunction of queries. Thus, conjunctions of interactive constraints can be easily translated into queries of a query language: the conjunction of constraints is intended as a tuple selection on a database, and the set of involved variables in the store as the tuple projection. This is subject of current investigation.

19.0 Related Work

The general idea that in many applications it is both unreasonable and unrealistic to force the user to provide in advance all information required to solve the problem was argued by Sergot [105] who proposed an extension of Prolog allowing interaction with the user.

The need for partial knowledge about domains has been underlined in [81] for configuration problems. Since configuration is a generative process, the domains are not known when creating variables because new possible components are generated during search. The employed solution, in this case, is a domain containing a *wildcard* element, that represents components that have not been generated yet; when new elements are known they are taken into account and considered into the domains. In our formulation, the unknown domain part is not considered as an element in the domain, but as a set of elements to be acquired. We think that in the ICSP formulation there is a more clear distinction between set membership and set inclusion. In fact, in [81], the wildcard element represents a set of elements, but is also an element of the domain. Our representation allows to handle domains in a declarative way. Our extension of the ECL^{i}PS^e finite domain library can be implemented without imperative side-effects. For instance, if a domain is $D = \{1|U\}$ and a new element (say 2) has to be added, we can declaratively state $U = \{2|U'\}$, automatically obtaining that $D = \{1, 2|U'\}$. In the formulation given in [81], to insert in a domain (() 1 *wildcard*) the value 2, the domain is imperatively changed into (()1 2 *wildcard*). In other words, the *wildcard* is a constant value that can be imperatively changed into a set of values, while the unknown domain part is a logical variable, that allows logical inferences. Thanks to this logical reading, if more expressive power is necessary, the library can be upgraded without changing its interface. For instance, if we want to allow domains to contain sets, we will need to change the implementation, but not the interface of the library. A domain in [81] cannot contain set elements and sets cannot be nested. Since the distinction between set membership and set inclusion is not clear, a modification of the interface should be done. From the operational viewpoint, our work minimizes the extraction of unknown elements, because in recognition processes this phase is computationally expensive; in [81] there is no interaction between subsystem, thus the cost of element generation is less pressing. Finally, our work deals with interaction between subsystems, and considers domains as communication channels between different software modules.

The notion of interactive domains has some similarities with the concept of streams [1] widely used for communication purposes in Concurrent Prolog. A stream is a communication channel which can be assigned to a term that contains a message and an additional variable. From this perspective, *interactive domains* represent communication channels between the producer of domain values and the constraint solver. The unknown part of the domain contains a message (a set of retrieved values) and an additional variable for future acquisitions.

In [121] a system is shown where domains are represented by means of streams, and constraints filter domain values. This system shows the effectiveness of the proposed idea, but does not consider interaction between subsystems. The algorithm uses a lazy evaluation of domains, in fact when the stream is accessed, it reconstructs only the part of itself that the program requires. Elements are first generated and then filtered by constraints. In our work, the further concept of interactive constraint is given, and constraints are passed to the generation system that actively uses them to retrieve only consistent elements. The effect is similar to the filtering of elements *before* their generation. The algorithm has been refined, independently, in [23], where a precise description of the backtracking phase is given, and it has been deeply studied in [3]. Also, a lazy version of Arc-Consistency has been proposed [103]. We believe that inserting the concept of value acquisition can yield even greater advantages.

In [37], an extension of the CSP model called Constraint Acquisition and Satisfaction Problem (CASP) is proposed. In a CASP, partial information is given about the constraints, while variables and domains are fully known at the beginning of the computation. After a solution is found by the constraint solver agent, the result is given to another agent (which can possibly be human) that decides if the solution is satisfactory and, if it is not, more constraints are imposed and a new solution requested. The ICSP differs from a CASP in that constraints are fully known, while domain values can possibly be unknown at the beginning of the computation and can be discovered during search.

Dynamic Constraint Satisfaction (DCS) [22] is a field of AI taking into account dynamic changes of the constraint store such as the addition, deletion of values and constraints. The difference between DCS and our approach concerns the way of handling these changes. DCS approaches propagate constraints as if they work in a *closed world*. Basically, in a DCS you can add or remove a constraint; thus you can also add and delete domain elements provided that they are all known from the beginning. In an ICSP, instead, domain elements that are

unknown can be requested and inserted in the domain. DCS solvers record the dependencies between constraints and the corresponding propagation in proper data structures [104] so as to tackle modifications of the constraint store as soon as new data changes. In this perspective, we also cope with changes since the acquisition of new values can be seen as a modification of the constraint store. However, we work in an *open world* where domains are left *opened* thanks to their unknown part. Unknown domain parts represent intensionally future acquisitions, i.e., future changes.

Another DCS framework was given for configuration problems [83]. This framework considers dynamicity in the set of variables; variables are introduced or removed during search by means of constraints. The aim is finding a solution where only some of the variables are assigned a value, while the others are *inactive*. However, the set of domain values is given at the specification of the problem.

Other related approaches concern constraint-based reactive systems [38]. Reactive programs are programs that react with their environment, are usually stateful, and have to take decisions before their consequences are known. They interleave information acquisition and decision making activities. Also, in our approach data acquisition and its processing are interleaved. However, in reactive programs the constraint solver computes a solution for a given set of (incrementally added) variables starting from an already committed system state. Thus, the computation of a single query is performed with no interaction with the producer of constrained data. In our approach, instead, these interactions take place during the execution of a single query.

As a final remark, in the field of programming languages, concurrent constraint programming [100] represents a framework which is based on the notion of consistency and entailment for dealing with partial information. Computation emerges from the interaction of concurrently executing agents communicating by means of constraints on shared variables. In a sense, the work on concurrent constraint programming focuses on the algorithms without paying attention to the semantics of the external world. With our approach we add this semantics. Concurrent constraint programming can be used as an effective language for implementing the interactive constraint solver.

PART V

A CLP Language with Interactive Constraint Satisfaction

20.0 Introduction

Combinatorial problems are often very hard to solve, and most of them belong to the NP-complete class. Constraint Logic Programming [69] is a declarative programming paradigm that enables the user to separate the problem description from the resolution algorithm. *"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."* [35]. In particular, CLP(FD) languages are widely used for various types of combinatorial problems, like scheduling, planning, visual recognition, timetable assignment and so on.

One drawback of CLP(FD) is that all the possible information about domains must be available at the beginning of the computation. Domains are internal data structures, where it is impossible to insert new elements once the constraint propagation is started.

In the previous chapter, we proposed the ICSP model as an extension of the CSP model. In an ICSP, variables range on partially known domains which have a known part and an unknown part represented as a variable. Domain values are provided by an extraction module and the acquisition process is (possibly) driven by constraints. The ICSP model has been successfully applied to visual recognition tasks [19] and planning [5]. It could be used to acquire domain values from the web (*"The interactive behavior of our constraint reasoner also can be seen as a form of ICSP"* [73]). Experiments on randomly-generated problems [21] show that in many instances, ICSP solving algorithms can avoid domain element acquisition steps, providing good performance in problems where information is partially specified.

In this chapter, we propose a corresponding CLP(FD) language where domains are left open and considered as *communication channels*. New elements can be inserted in domains and evaluated lazily by constraints. The operational semantics is based on the new concept of *known consistency*, where constraints are checked against known values, while newly acquired values are processed lazily. The propagation engine follows the ideas developed for Interactive Constraint Satisfaction Problems (ICSP) [20]. In the ICSP model, domains can be variables, values are acquired during the solution process only when needed, and inserted into variable domains. Moreover, constraints can be used to guide an acquisition module for extracting possibly consistent values.

21.0 Syntax and Declarative Semantics

The language we consider lets the user work with unbound domains, interactively acquire new information and impose constraints on domains. The proposed language is a two-sorted CLP language, where the sorts are the classical *FD* sort on finite domains and a sort based on information channels [44, 45, 47]. We first define constraints and operations on the sorts, then we link the two sorts.

In the following, we will comply to the conventions in [69]. In particular, every constraint domain $\mathcal{C}$ (where $\mathcal{C}$ can be FD, $\mathcal{P}_{FD}$ or the $FD{+}\mathcal{P}_{FD}$ sort) contains: the constraint domain signature $\Sigma_{\mathcal{C}}$, the class of constraints $\mathcal{L}_{\mathcal{C}}$, the domain of computation $\mathcal{D}_{\mathcal{C}}$, the constraint theory $\mathcal{T}_{\mathcal{C}}$, the solver $solv_{\mathcal{C}}$.

21.1 The FD sort

The FD sort contains the usual constraints in CLP(FD) languages (arithmetic, relational constraints plus user-defined constraints). We suppose that the symbols $<, \leq, +, -, \times, \ldots$ belong to Σ_{FD} and are interpreted as usual.

21.2 CLP(S-Channels)

The aim of the second sort is providing communication channels as first class objects. In a sense, they can be considered as *streams*, but they are intrinsically non-ordered, and do not contain repeated elements. We call the sort S-Channels (Set-Channels). CLP($\mathcal{SET}$) [26] and CLP(S-Channel) differ in the operational semantics and in the language expressivity. Nonground elements are forbidden in an S-Channel; this restriction allows for more efficient algorithms for propagation.

In CLP(S-Channel), the constraint domain signature, $\Sigma_{S-Channel}$, contains the following constraints:

- $s\text{-}member(E, S) \Leftrightarrow E \in S$
- $union(A, B, C) \Leftrightarrow A \cup B = C$
- $intersection(A, B, C) \Leftrightarrow A \cap B = C$

- *difference*$(A, B, C) \Leftrightarrow A \setminus B = C$
- *inclusion*$(A, B) \Leftrightarrow A \subseteq B$

21.3 Linking the two sorts

Intuitively, we want to bind a CLP(FD) system with a CLP(S-Channel) system with the intended semantics that variable objects in the S-Channel environment provide domains for the variables in the FD world.

Given the two CLP languages $\mathcal{L}_{FD}$ and $\mathcal{L}_{S-Channel}$, we define the CLP language $\mathcal{L}$ as the union of the two languages, with a further constraint :: defined as follows:

- the signature $\Sigma = \Sigma_{FD} \cup \Sigma_{S-Channel} \cup \{ :: \}$;
- the intended interpretation $\mathcal{D}$ keeps the original mappings in the FD and *S-Channel* sorts; i.e., $\mathcal{D}|_{\Sigma_{FD}} = \mathcal{D}_{FD}$ and $\mathcal{D}|_{\Sigma_{S-Channel}} = \mathcal{D}_{S-Channel}$.

The declarative semantics of the constraint is

$$X :: S \Leftrightarrow X \in S$$

The symbol :: is overloaded, and represents the usual CLP(FD) unary constraint linking a domain variable X to its domain when S is ground and the binary constraint of set membership when S is a channel. From the S-Channel viewpoint, it is just a set membership operator; its power comes from its operational semantics.

21.4 Example: A vision system

With the described language, defining the model of a vision system is quite straightforward. We can define the constraints as in the model given in Section 8:

rectangle(X_1, X_2, X_3, X_4) :-
touch(X_1, X_2), *touch*(X_2, X_3), *touch*(X_3, X_4), *touch*(X_4, X_1),
no_touch(X_2, X_4), *no_touch*(X_1, X_3),
same_length(X_2, X_4), *same_length*(X_1, X_3),
perpendicular(X_1, X_2), *perpendicular*(X_2, X_3),

$perpendicular(X_3, X_4)$, $perpendicular(X_4, X_1)$,
$parallel(X_2, X_4)$, $parallel(X_1, X_3)$.

In our framework, we can state that the variables range over the set of segments in the image, which is unknown at the beginning of the computation:

$$X_1, X_2, X_3, X_4 :: Segments$$

Then, each time a segment s_i is retrieved, it will be inserted in the channel $Segments$ over which the FD variables range, by simply stating $s\text{-}member(s_i, Segments)$.

Moreover, if we need to extend the model and make it more selective, we could add more constraints or consider more different types of visual features. The language supports modification of the model, allowing fast prototyping and easy modification. Since it is a CLP system, constraints can be easily added to the model; moreover, we can easily add different types of visual features, possibly retrieved by different segmentation agents by defining more Set-Channels.

For example, we could state that a group of variables range over the set of surface patches, another over the set of segments, another over the set of angles, simply by stating:

$$\begin{array}{rcl} S_1, S_2, \ldots, S_i & :: & Segments \\ P_1, P_2, \ldots, P_j & :: & Surfaces \\ A_1, A_2, \ldots, A_k & :: & Angles \end{array}$$

In this way, different acquisition modules can be exploited at the same time, providing in parallel different classes of information. It is worth noting that even in human vision there are different brain parts that carry on recognition of different classes of features (e.g., horizontal and vertical lines). Moreover, from a certain viewpoint a surface may appear as a segment; in this case we can state that some variable can range on the union of the two sets:

$$S_x :: D, union(Segments, Surfaces, D).$$

If we have two cameras, we can exploit stereo-vision to infer distance information about the various objects. Of course, the two cameras will see only some objects in common, so a variable could range only on the set of objects in the intersection:

$$S_{3D} :: D_{3D}, intersection(Camera_L, Camera_R, D_{3D}).$$

21.5 Generalization

We can generalize the approach and consider a CLP($FD + \mathcal{P}_{FD}$) sort, in which the domains can be any collection of FD elements.

From a language viewpoint, we want the employed CLP($\mathcal{P}_{FD}$) language to provide (as built-ins or user-defined constraints) the constraints given in section 21.2. Other requirements come from the operational semantics.

Our framework can be implemented on top of different CLP languages, dealing with different domain formalizations, thus achieving different expressive power and efficiency levels. Besides channels, we could build a two-sorted CLP system on top of a CLP dealing with *lists* [17] or *sets* [26, 50]. Of course, the obtained system inherits advantages and drawbacks from the host language. For example, sets are managed in a wide variety of ways, so, if we have a combinatorial problem and need a very efficient system, we can choose Conjunto [50], while if we need higher expressive power, we may prefer $\{log\}$ [25].

22.0 Operational Semantics

The operational semantics of the proposed framework is defined by the propagation of constraints in the two sorts, plus the propagation of the constraint :: that links the sorts.

22.1 Operational Semantics: CLP(S-Channels)

In the domain sort, objects in the domain of computation are channels.

Declaratively, the semantics of the S-Channel sort is the same of a Set sort. Operationally, we want to achieve high efficiency, thus we need to avoid non-deterministic unification [25]. We restrict the language by forbidding non ground terms to enter an S-Channels.

An S-Channel is internally represented as a difference list; i.e., it is based on a data structure S defined as:

$$\begin{array}{lll} S & ::= & \{\} \\ S & ::= & \{T|S\} \\ S & ::= & V \end{array}$$

where T is a ground term and V is a variable. The structure S can be considered split into a known part (containing all the ground elements) plus an unknown part (the eventual variable representing the tail of the channel). If the unknown part is empty, then the channel is called *closed*, otherwise it is *open*. The internal data structure is hidden to the user, that can only use the constraints and operations defined in Section 21.2. The two parts are kept disjoint for efficiency reasons. Constraints access an S-Channel through the following primitives (Fig. 22.1):

- *insert(S,E)*: inserts a ground element E in the channel S. If S is open and E was not previously inserted, this primitive adds E to S. If S is closed, then *insert* checks if E is in S: if $E \in S$ the primitive succeeds, otherwise it fails.
- *get(S,E)*: unifies E with an element that was previously inserted in S. If no element in S can be unified to E and S is open, it suspends. If no element can be unified and S is closed, then it fails.
- *known(S,L)*: provides the list of known elements in S.
- *close(S)*: states that no more elements belong to the channel.

```
insert(S,E) :- nonground(E), !, raise_exception.
insert(S,E) :- var(S), !, S={E|_}.
insert({},E) :- !, fail.
insert({E|_},E) :- !, true.
insert({_|T},E) :- insert(T,E).

get(S,E) :- var(S), !, suspend.
get({},_) :- !, fail.
get({E|_},E) :- !.
get({_|T},E) :- get(T,E).

known(S,[]) :- var(S), !.
known({},[]).
known({E|T},[E|T1]) :- known(T,T1).

close(S) :- var(S), !, S={}.
close({}).
close({_|T}) :- close(T).
```

Figure 22.1 Primitives

All the other operations, and constraints in particular, are based on the primitive operations; for example, *s-member(E,S)* works either as an *insert* or a *get* operation. If the element E is ground, then it is inserted in the channel S. If the element is not ground, then it is nondeterministically bound to an element that was previously inserted in the channel. Considering the operational behavior of the primitives, *s-member*: (i) tries to insert E in S if E is ground (ii) tries to unify E to one element in S if E is not ground (iii) suspends otherwise. *union*(A, B, C) checks that each element in A or in B also belongs to C. Moreover, it propagates information about closed channels: for instance, if A and C are closed, then all the elements that belong to $C \setminus A$ are inserted in B.

No constraint performs nondeterministic operations, in order to be more efficient. It suspends whenever more than one alternative choice is possible.

22.2 Operational Semantics: linking the two sorts

In CLP(FD) and in CLP($\mathcal{P}_{FD}$), domains are operationally updated in different ways. In CLP(FD) a domain is simply the collection of values that a variable can take, and, in efficient CLP(FD) systems [24, 2, 14], domains are updated with non logic, imperative

primitives. Elements that cannot be part of a consistent solution are removed from the domain, thus pruning the search space. On the other hand, in CLP($\mathcal{P}_{FD}$), a domain should be obtained as a logic consequence of the theory and the program; thus no imperative side-effects can be allowed.

For example, consider the constraint $D_1 \supseteq \{1, 2\}$ stating that D_1 must contain at least two values. D_1 could be also the domain of an FD variable X_1, i.e., $X_1 :: D_1$ and there could be a constraint stating that $X_1 \neq 2$. Usual CLP(FD) propagation of $X_1 \neq 2$ would remove value 2 from the domain of X_1; this elimination is inconsistent with $D_1 \supseteq \{1, 2\}$, so we would have a failure. This failure is not correct, in fact the set of constraints $(D_1 \supseteq \{1, 2\})$ $\wedge$ $(X_1 :: D_1) \wedge (X_1 \neq 2)$ is satisfiable (declaratively, :: means set membership). On the FD side we wish to delete the inconsistent element from the domain, while on the $\mathcal{P}_{FD}$ side we should not delete elements, because the domain is also a logical variable.

Thus, we have a double representation of domains: one keeps a clear declarative reading, while the other is hidden to the user and is used to perform efficient, FD propagation. Each domain D_i is represented de facto by two structures:

The definition domain D_i^d It is a set variable accessible at language level, thus, it can be defined intensionally by constraints. Its state can be modified exclusively by constraints imposed on it, never by constraints imposed on the X_i variable ranging on D_i.

The current domain, D_i^c It represents the standard FD variable domain, i.e., it is the set of (possibly consistent) values the variable X_i can assume. It is hidden to the user and the values it contains are limited by constraints on the FD and on the $\mathcal{P}_{FD}$ sorts.

Current domains should provide all the available information from the $\mathcal{P}_{FD}$ sort to the FD sort, while giving the possibility to delete inconsistent values. The internal representation of a current domain inherits from the representation of definition domains (i.e., channels in the $\mathcal{P}_{FD}$ sort). Current domains can be considered as split into a known part and an unknown part, like definition domains.

For each domain D_i, the basic property $D_i^c \subseteq D_i^d$ is maintained true. With this representation, inconsistent elements can be deleted from the current domain of a variable without affecting the definition domain. D_i^c must synthesize all the knowledge provided by the definition domain; in particular, the known elements and the (eventual) emptiness of the unknown part. All the elements in the known part of the definition domain should be also

in the known part of the current domain in order to be exploited for propagation. Knowledge on the unknown domain part is important because we can exploit it for constraint propagation. In fact, a domain element $v \in K_i$ can be removed by a binary constraint $c(X_i, X_j)$ only if we know that it is not supported by any element in the domain of X_j; as any element could be supported by unknown elements, we cannot delete it if the domain of X_i is not fully known (i.e., not closed).

Declaratively, constraints limit the combination of assignments of elements taken both from the known domain part and from the unknown domain part. Operationally, they delete from the known domain part elements that cannot lead to a solution and filter elements in the unknown domain part. For this reason, we impose *auxiliary constraints* on the unknown domain part that provide information on the set of elements it should contain.

For example, a constraint $c(X_i, X_j)$ deletes inconsistent elements from the known domain part and imposes the auxiliary constraints $c^{u_j}(X_i, U_j^c)$ and $c^{u_i}(U_i^c, X_j)$. A constraint like $c^{u_j}(X_i, U_j^c)$ is a relation between an object and a set of objects, linking every domain element with a subset of the second domain. For instance, a constraint $X > 5$, where $X :: D$ and $D = \{1, 10|U\}$, produces the new domain $\{10|U'\}$ with the imposed constraint $U' >^{\in} 5$, i.e., $\forall v \in U', v > 5$.

An auxiliary constraint can only be imposed by another constraint (it is not accessible at the language level) and its purpose is twofold. First, it prohibits inconsistent elements from entering the domain, i.e., whenever new information is available, it checks the synthesized elements, and eventually deletes them. Second, it can be exploited to provide information on consistent elements, and to help the synthesis of consistent elements[1].

Definition 22.1. *A constraint $c(X_i, X_j)$ defines two* auxiliary constraints *$c^{u_i}(U_i^c, X_j)$ and $c^{u_j}(X_i, U_j^c)$. $c^{u_i}(U_i^c, X_j)$ is a subset of the cartesian product $\mathcal{P}(D_i^c) \times D_j^c$ representing couples of compatible assignments of the subdomain $U_i^c \subseteq D_i^c$ and the variable $X_i \in D_j^c$. It is satisfied by a couple of assignments $U_i^c = S_i \subseteq D_i^c$, $X_j = v_j$ iff $\forall v_i \in S_i, (v_i, v_j) \in c(X_i, X_j)$*

Finally, a predicate `promote` should be exported by the $\mathcal{P}_{FD}$ module (i.e., it should be implemented using $\mathcal{P}_{FD}$ structures). The `promote` predicate should ideally move elements from the unknown domain part to the known domain part (or fail if the unknown part is

[1]This possibility can provide a huge performance improvement, but it can be exploited only if the desired result does not contain definition domains. In fact, the synthesis of elements with auxiliary constraints has an implicit influence on the $\mathcal{P}_{FD}$ computation.

empty). In the S-Channel sort it is implemented with an `insert` operation. The motivations for the `promote` predicate will be given in the following section.

22.3 Operational Semantics: Finite Domains

We propose an extension for the partially known case, of the concept of consistency, called *known consistency*. In this work, we provide only the extension of node-consistency and arc-consistency; the extension to higher degrees of consistency is straightforward.

Definition 22.2. *A unary constraint $c(X_i)$ is* known node-consistent *iff*

$$\forall v_i \in K_i^c,\ v_i \in c(X_i)$$

Definition 22.3. *A binary constraint $c(X_i, X_j)$ is* known arc-consistent *iff*

$$\forall v_i \in K_i^c,\ \exists v_j \in K_j^c \ s.t.\ (v_i, v_j) \in c(X_i, X_j)$$

Definition 22.4. *A constraint network is* known arc-consistent (KAC) *iff all unary constraints are known node-consistent and all binary constraints are known arc-consistent.*

Lemma 22.3.1. *If a network of constraints is KAC, then the subset of known elements is an arc-consistent subdomain.*

Proof. From definition 22.4, for each constraint $c(X_i, X_j)$ each known element in D_i^c has a support in D_j^c. □

Achieving arc-consistency means translating a given problem P into a different but equivalent problem P' in which the constraint network is arc-consistent. Many algorithms have been given for computing arc-consistency (named AC-2 [116], AC-3 [80], AC-4 [84], AC-5 [61], AC-6 [6] and AC-7 [7]); the algorithms have different performance, but all are detect inconsistency in the same problem instances (when a domain becomes empty). In the same way, there are many algorithms that translate a problem, P, into an equivalent problem P', with different domains, such that the new problem P' is KAC.

Proposition 22.3.2. *Every algorithm achieving KAC (i.e. any algorithm that computes an equivalent problem that is KAC) and that ensures at least a known element in each variable domain is able to detect inconsistency in the same instances as an algorithm achieving AC.*

Proof. An algorithm achieving KAC computes an arc-consistent subdomain (lemma 22.3.1). Since each domain must contain at least one (known) element, then the obtained known subdomain is a not empty subset of the maximal arc-consistent subdomain. □

In other words, if an arc-consistent subdomain exists, then a maximal arc-consistent subdomain exists; so if KAC does not detect inconsistency, AC will not detect inconsistency either. The advantage in using KAC is that the check for known arc-consistency can be performed lazily, without full knowledge of all the elements in every domain.

Operationally, achieving KAC has some similarities with achieving *Lazy Arc Consistency* (LAC) [103]. LAC is an algorithm that finds an arc-consistent subdomain (not necessarily a maximal one) and tries to avoid the check for consistency of all the elements in every domain. KAC looks for an arc-consistent subdomain as well, but it is aimed at avoiding unnecessary information retrieval, rather than unnecessary constraint checks.

In order to achieve AC, algorithms need to remove elements that are proven to be inconsistent. In order to achieve KAC, algorithms need to delete elements and to *promote* elements, i.e., to move ideally some elements from the unknown part to the known part. Of course, in order to obtain a problem equivalent to the original one, only elements that satisfy all the auxiliary constraints can be promoted. Elements can then be deleted if they are shown to be inconsistent. In figure 22.2 an algorithm for achieving KAC is shown, based on AC-3 as presented in [80]; other algorithms could be employed as well. When all domains are fully known from the beginning of the computation, then procedure KAC provides maximal subdomains, i.e., it computes AC.

`promote(`v`,`U^c`)` is a predicate that returns *true* if there is an element v in U^c that can be promoted and *false* otherwise. Implementation of the `promote` predicate in each setting is a crucial issue, because it highly influences efficiency. Heuristics can be exploited to select the most promising elements to be promoted; in particular, auxiliary constraints can be used to drive the provider component. For example, in a visual search problem [19], auxiliary constraints can be passed to an acquisition module that extracts visual features (such as segments or surfaces) from an image. Auxiliary constraints can provide information to the extraction agent, and focus its attention on the most significant image parts.

```
procedure KAC
Q ← {c(X_i, X_j)};
while not empty(Q)
    select and delete one constraint c(X_k, X_m) from Q;
    if REVISE(c(X_k, X_m)) then
        Q ← ⋃{c(X_i, X_k) such that i ≠ k ∧ i ≠ m}
    EndIf;
EndWhile;
End KAC

function REVISE(X_i, X_j):bool;
MODIFIED ← false;
for each v_i ∈ K_i^c do
    if there is no v_j ∈ K_j^c such that (v_i, v_j) ∈ c(X_i, X_j)
    then
        MODIFIED ← true;
        if promote(v, U_j^c)
            then U_j^c = {v|(U_j^c)'}
            else delete v_i from K_i^c;
        EndIf;
    EndIf;
EndFor;
return MODIFIED;
End REVISE
```

Figure 22.2 KAC propagation for FD constraints

22.4 Operational Semantics: $\mathcal{P}_{FD}$ sort

In this section, we generalize the approach; besides the S-Channel sort, we show that other $\mathcal{P}_{FD}$ systems can be employed for the CLP(FD+$\mathcal{P}_{FD}$) framework. We define properties and behaviors that the sort on domains should provide in order to be exploited in our framework. Then, we show that two utterly different systems (Conjunto and {log}) can give the requested functionalities. We could thus obtain some interesting variants: Lazy Conjunto, (i.e, using Known Arc-Consistency in Conjunto), or CLP(FD)+{log} (i.e., FD propagation in {log}).

Since the scope of our computation is finding solutions to a CSP, we need to distinguish domain elements that can be fruitfully exploited in a CSP computation from elements that are not yet known. For this reason, the employed system should allow the partitioning of

each domain D_i^d into a known and an unknown part (respectively, K_i^d and U_i^d), with the intended meaning that the known part contains all the ground elements, while the unknown part is the rest.

The following properties should be provided by a CLP($\mathcal{P}_{FD}$) system to be included in the CLP(FD+$\mathcal{P}_{FD}$) framework:

Property 1. Partitioning domains into a known and an unknown part: *for each domain D_i^d, at every step of the computation, the set of (ground) elements that are proven to belong to the set can be distinguished. We call this set the* known domain part K_i^d. *The rest of the domain (i.e., the set $U_i^D = D_i^d \setminus K_i^d$) is called the* unknown domain part

- In {log} domains are given by an arbitrary collection of Prolog terms; sets are built using the set constructor $\{.|.\}$ plus the empty set $\emptyset$. For example, $\{1, p(2), Z, q(Y,3)|K\}$ is a valid set. {log} satisfies Property 1; the known part is the set of ground elements that are proven to belong to a set, in our example, $\{1, p(2)\}$. Also, each time a new element is proven to belong to the known part, we need to impose a constraint stating that the element must not belong to the new unknown part. For example, if we have a domain containing $D_i = \{p(a,X), p(Y,b)\}$ (thus $K_i^d = \emptyset$), and we prove that $p(a,b) \in D_i^d$ we have $(K_i^d)' = \{p(a,b)\}$ and $(U_i^d)' = \{p(a,X), p(Y,b)\}$; we will have thus to impose an explicit constraint stating $p(a,b) \notin (U_i^d)'$.

- In Conjunto a set S is represented by two ground sets: a Least Upper Bound (LUB) and a Greatest Lower Bound (GLB), such that $GLB_S \subseteq S \subseteq LUB_S$. Conjunto satisfies Property 1 and the known part coincides with the GLB. The unknown domain part[2] is simply given by the set difference $U_i^d = D_i^d \setminus K_i^d = LUB \setminus GLB$.

Property 2. Promotion of elements: *It is possible to define a predicate* `promote` *which can select an element from the unknown domain part and synthesize or acquire enough information to let it enter the known domain part.*

The `promote` predicate can be implemented in a variety of ways, depending on the problem and on the $\mathcal{P}_{FD}$ sort we are exploiting. In general, each computation that provides enough information about domain elements can be exploited in this predicate (i.e., each computation that synthesizes a ground element can be used to implement the `promote` predicate).

[2]In this case, the term *unknown* is not well-suited, because in Conjunto all the domain elements have to be known. We use it here to keep the same symbols of the rest of the thesis.

However, since propagation will try to minimize the number of calls to `promote`, it is more convenient to associate a computation performing time-consuming operations. In the *channel* sort, i.e., when domains are communication channels between subsystems, the promotion is a request for a domain element to the value provider. For instance, it can be used for human interaction (the user declares the domain elements) or for acquisition of information from another component.

Property 3. Test/Assertion of emptiness*: the* `promote` *predicate has to provide a 'null value' information if there are no more elements in the unknown domain part to be promoted. In other words,* `promote` *must be a predicate reporting* false *if there are no more consistent values.*

This property is very important, as we need a way to test if the unknown part is empty.

22.5 Example: Numeric CSP

Let us see with a simple example, how a sample problem can be described and solved in CLP(FD+$\mathcal{P}_{FD}$). With the given language, we can state in a natural way the following problem:

$$:\!-X :: D_X, Y :: D_Y, Z :: D_Z, union(D_X, D_Y, D_Z), Z > X.$$

defining three variables, X, Y and Z, with constraints on them and their domains. KAC propagation can start even with domains fully unknown, i.e., when D_X, D_Y and D_Z are variables. Let us suppose that the elements are acquired through interaction with a user (i.e., the $\mathcal{P}_{FD}$ sort is based on channels and the `promote` predicate is specialized for acquisition from a user) and the first element retrieved for X is 1. This element is inserted in the domain of X, i.e., $D_X^d = \{1|(D_X^d)'\}$; since it is consistent with auxiliary constraints, it is also inserted in the current domain, i.e., $D_X^c = \{1|(D_X^c)'\}$. Then KAC propagation tries to find a support for this element in each domain of those variables linked by FD constraints; in our instance D_Z. A value is requested for D_Z (eventually, providing the user the imposed auxiliary constraints in order to have guided acquisition) and the user gives a (possibly consistent) value: 2 (Figure 22.3). This element is inserted into the domain of Z: $D_Z^d = \{2|(D_Z^d)'\}$. Since the acquired element is consistent with auxiliary constraints, it is not deleted and it is inserted in the current domain of Z. Finally, the constraint imposed on domains can propagate, so element 2 has to be inserted in both the (definition and current) domains of Y and X, thus $D_Y^c = D_Y^d = \{2|D_Y'\}$ and $D_X^c = D_X^d = \{1, 2|D_X''\}$.

```
:- X :: Dx, Z :: Dz, intersection(Dx,Dy,Dz), Z > X.
List new values for X: {1|Dx'}.
The following constraint should guide
acquisition:  Z>1
List new values for Z: {2|Dz'}.
The following constraint should guide
acquisition:  Z>2
List new values for X: {}.

Dx={1, 2 | _}, Dy={2 | Dy'}, Dz={2 | Dz'}
X::{1 | _}, Y::{2 | _}, Z::{2 | _}

delayed goals:
intersection(Dx,Dy,Dz)
Z>X

yes.
```

Figure 22.3 Example of computation

KAC propagation must now find a known support for element 2 in D_X^c, so another request is performed. If the user replies that there is no consistent element in the domain of Z, then 2 is removed from the current domain of X. It will remain in the definition domain because the element semantically belongs to the domain even if no consistent solution can exist containing it.

When KAC propagation reaches the quiescence, in each domain we know a support for each other known value.

23.0 Implementation Details

We implemented our framework on the ECL^{i}PS^e [2] CLP system; the architecture of the implementation is depicted in Figure 23.1. FD constraints impose auxiliary constraints. In current implementation, only unary auxiliary constraints are allowed, because we suppose that the acquisition system providing values taken from the outer world is able only to check unary constraints. If the acquisition agent is more powerful, the architecture can be simply extended. Auxiliary constraints are passed to the acquisition agent and can be exploited to provide possibly consistent values. Each retrieved value belongs semantically to the definition domain of the considered variable, thus it is inserted. Propagation of the :: constraint provides values from the definition domain to the current domain; this value passing is filtered by auxiliary constraints. Other elements can be deleted by constraints imposed on the FD side.

In our system, a value can be deleted because of FD propagation, or it can be inserted in the known domain part by propagation of the :: constraint. For this reason, each possible domain value can be in one of the following states:

- *present*: the value is proven to belong to the domain (i.e., belongs to the known domain part)
- *deleted*: the value used to belong to the domain, but has been removed because of constraint propagation
- *unknown*: during computation, it has never been shown that the value either belongs or does not belong to the domain (i.e., it semantically belongs to the unknown domain part).

During unification, for each value in the first domain, the status of the same element in the second domain will be changed accordingly. The only allowed transitions are from the state unknown to present to deleted; in this way, a domain value cannot be deleted and then re-inserted into a domain (thus leading to unstable behavior).

23.1 Related work

S-Channels can be considered as streams with a set semantics (i.e., in an S-Channel there are no repeated elements and elements are not sorted). Streams are widely used for

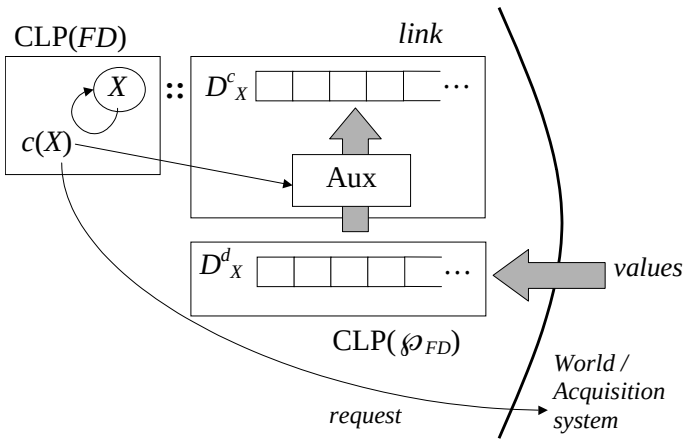

Figure 23.1 Architecture

communication purposes in concurrent logic programming [106]. Various communication protocols can be obtained through this simple yet powerful data structure [107]. In an S-Channel only ground elements are allowed; this forbids (open) channels to be passed in a channel, but allows higher efficiency. If we allow non-ground elements to be in an S-Channel, we obtain a more expressive framework, like {log}; thus if more expressive power is necessary, another instance of the CLP(FD+$\mathcal{P}_{FD}$) framework, the CLP(FD)+{log} instance, can be used.

Many systems consider set objects, because sets have powerful description capabilities. In particular, some have been described as instances of the general CLP($\mathcal{X}$) framework [69], like {log} [25, 27], CLPS [79], or Conjunto [50].

In {log} [25, 27], a set can be either the empty set $\emptyset$, or defined by a constructor `with` that, given a set S and an element e, returns the set composed of $S \cup \{e\}$. This language is very powerful, allowing sets and variables to belong to sets. However, the resulting unification algorithm is non-deterministic; in the worst case, it has an exponential time complexity.

Conjunto [50] is a constraint logic programming language in which set variables can range on set domains. Each domain is represented by its greatest lower bound and its least upper bound. Each element in a set must be ground, sets are finite, and they cannot contain sets. These restrictions avoid the non-deterministic unification algorithm, and allow good performance results. (Conjunto is a library of the language ECL^{i}PS^e [2]). However, Conjunto does not deal with problems in which some domain elements are not known; in

fact, the user must provide the least upper bound of each set, thus giving the universe set at the beginning of the computation.

Set variables are based on a similar idea: as in Conjunto, each variable has a Least Upper Bound and a Greatest Lower Bound. Set variables are embedded in many constraint programming languages, like SICStus Prolog [14], ECL^{i}PS^e [2] and ILOG solver [91].

In [16] a method for compiling constraints in CLP(FD) is described. There is only one primitive constraint X `in` R, used to implement all the other constraints. R represents a collection of objects and can also be a user function. Thus, in `clp(FD)` domains are managed as first-class objects; our framework can be fruitfully implemented in systems exploiting this idea, like SICStus Prolog [14].

In [105] a framework is proposed to deal with interaction with the user in a logic programming environment. Our work can be used for interaction in a CLP framework; it lets the user interactively provide domain values.

Concurrent Constraint Programming [100] is a framework dealing with constraints and partial information. It is based on the notions of consistency and entailment of constraints imposed on shared variables. For these reasons, it would be a good framework for an implementation of CLP($FD + \mathcal{P}_{FD}$) in a concurrent environment.

As we discussed in Section 19, in [81] a system is shown that considers unbound domains. A domain can contain a "wildcard" element that can be linked to any possible element. Operationally, this idea has some similarities to the ICSP, however in the ICSP formulation there is a more clear distinction between set membership and set inclusion. In fact, in [81], the wildcard element represents a set of elements, but is also an element of the domain. Since our formulation has a more clear declarative semantics, the ICSP model was extended to interface with sets, and ICSP algorithms were used to implement constraint propagation in CLP(FD+$\mathcal{P}_{FD}$).

PART VI

Modeling 3D Objects

24.0 View Centered and Object Centered models

When dealing with 3D object recognition, object pose must be considered. When seen from some viewpoints, some object features could be invisible, and others could be partially overlapped, thus showing a different shape. We have two possible ways of recognizing a 3D object: either by defining a set of single view graphs corresponding to alternative views of the 3D object (i.e., view-centered models), or by defining a single object model independent from the point of view (i.e., object-centered model).

In the first case, we recognize the object by performing a perfect matching of retrieved features and parts of a single view-centered model. In the second case, instead, we have to identify a sub-graph representing the object view and, in a sense, relax other constraints and accept solutions where not all variables are assigned a visible image feature.

The first idea enables objects to be easily recognized, since the same techniques used for 2D recognition can be used for 3D recognition without changes. Each object is recognized when its view perfectly matches one of the possible view models. However, the definition of multiple views is not easy and the user has to consider a-priori all possible view points, in order to guarantee the completeness of the recognition process. The modelling phase becomes thus very complex and tedious, mistakes become likely and long debugging sessions can be expected. Moreover, the problem is divided into a sequence of CSPs: one for each possible viewpoint. Each of the CSPs must be solved in order to check if the object is present in the scene in the particular pose.

The second idea allows the user to define one single model of the object to be recognized and leaves the system the duty to perform the matching. In a sense, in the first case the user has to provide his/her knowledge about the possible viewpoints. In the second case, the system has the knowledge about the possible viewpoints and the user has only to define the object model at a higher level. We consider a better solution the object centered model, where the user defines the object by means of intuitive properties and let the solver do the hard work. This idea fits well into the philosophy of Constraint Programming: "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [35].

In an object centered model, we know that not all the visual features can be seen at a same time. This problem can be addressed by considering a *Null value* in the domains:

each CSP variable can range on the set of features plus a value representing an invisible feature.

$$Vars \in Features \cup \{invisible\}$$

Of course, the *invisible* value must be taken into account when implementing constraints. We partition all the possible constraints involving features into two groups:

- *object constraints* are topological or geometric constraints describing the object itself, in a way completely independent from the viewpoint;
- *visual constraints* are constraints that inform the system about the appearance of the involved features.

In particular, object constraints will be always satisfied if one of the involved features is invisible. More formally:

$$c(X_i, X_j) = \begin{cases} TRUE & if \quad X_i = invisible \vee X_j = invisible \\ \langle verify\_cond(c, X_i, X_j) \rangle & if \quad X_i, X_j \in Features \end{cases}$$

In a sense, the behavior of object constraints in presence of invisible features can be considered as a constraint relaxation. It is worth noting that in a CLP environment, these constraints can nevertheless be kept in the constraint store and provided as *conditional answers*. This way, if we are able to obtain further information, we can test if our solution was correct. In other words, the operational semantics would be changed as:

$$c(X_i, X_j) = \begin{cases} Delay & if \quad X_i = invisible \vee X_j = invisible \\ \langle verify\_cond(c, X_i, X_j) \rangle & if \quad X_i, X_j \in Features \end{cases}$$

Note that, in this case, conditional answers provided when *invisible* values are encountered are particularly useful and interesting. In fact, if we are able to change the viewpoint of the camera, or obtain further information on the other side of the object, we can test if our identification was correct. Moreover, we inform the user about the (possible) appearance of the hidden side of the object. This resembles human vision: usually, when we see an object, we have some *expectations* about the appearance of the other side of the object.

24.1 Visual Constraints

Visual constraints can actively handle information provided by the *invisible* value in order to prune the search space. They provide information about appearance of features and exploit only information about appearance.

Visual constraints can provide mainly two types of information about surfaces:

- *partial occlusion information*
- *hiding information*

In other words, a surface can be (*i*) perfectly visible and perceivable by the segmentation agent (*ii*) *hidden*, i.e., not visible at all from the considered viewpoint, (*iii*) partially overlapped behind another surface, or very inclined, thus making impossible the exact recognition of the surface. With visual constraints, feasible views can be easily distinguished from infeasible ones.

Let us consider some simple examples of visual constraints; then we will generalize and show some properties.

Example The visual constraint $hide(X_i, X_j)$ states that at most one of the objects can be visible, while the other must be labelled *invisible*. Formally,

$$hide(X_i, X_j) = \begin{cases} TRUE & if \quad X_i = invisible \vee X_j = invisible \\ FALSE & if \quad X_i, X_j \in Features \end{cases}$$

This constraint is typically associated with opposite surfaces. In this way, we can immediately label *invisible* a surface if the opposite one is visible.

Example The visual constraint $overlap(X_i, X_j)$ informs the constraint solver that if X_i is visible, then X_j cannot probably be perceived correctly by the acquisition system. For this reason, the constraints involving its shape have to be relaxed, or, equivalently, the constraints involving X_j's shape are imposed only if X_i is shown to be invisible. In a sense, this can be considered a meta-constraint, providing information about the existence of other constraints:

$$overlap(X_i, X_j) \Longleftrightarrow X_i = invisible \vee c(X_j)$$

where $c(X_j)$ is the conjunction of all the unary constraints using information about the shape of X_j. Again, relaxed constraints can be provided as conditional answers, thus giving information about the expected shape of the partially hidden features.

The constraint *overlap* is usually associated with concave parts, where self-occlusion can be possible.

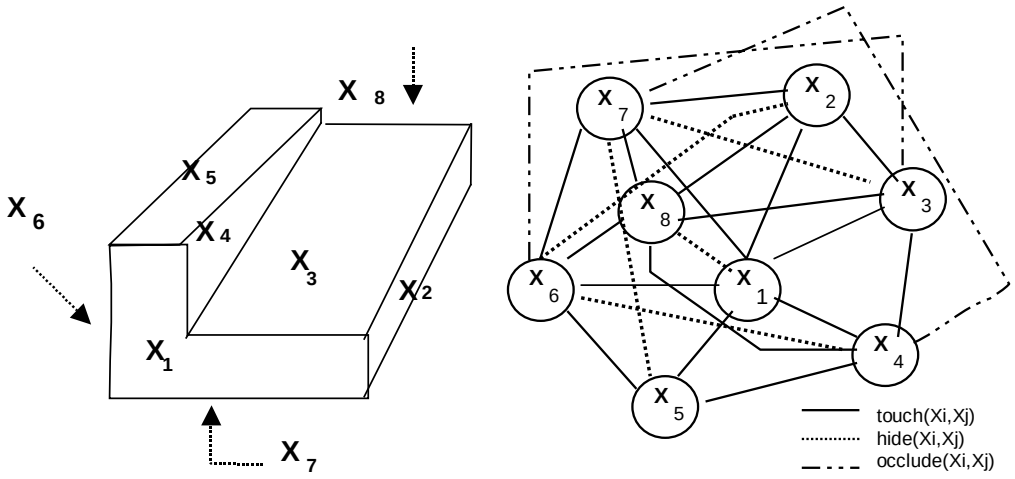

Figure 24.1 L-shaped 3D object model

For example, suppose that we are looking for an L-shaped object, with the model in Figure 24.1. For the sake of simplicity, we show only *touch* constraints, as an instance of object constraints, and *hide* and *occlude* constraints, as instances of visual constraints. Its corresponding graph has eight nodes $X1, \ldots, X8$, representing the eight object's surfaces. They are linked by topological constraints imposing that $X1$ touches $X2$, $X3$, $X4$, $X5$, $X6$ and $X7$, $X2$ touches $X1$, $X3$, $X7$ and $X8$ and so on. *Touch* constraints are indicated with a solid line in Fig. 24.1. We also know that if we see $X1$, we can never see $X8$ since they are opposite; for the same reason if we see $X6$ we can never see $X4$ and $X2$, and so on. These *hide* constraints are shown in Fig. 24.1 by dotted lines.

In Figure 24.2 we have an example of recognition. The segmentation module provided five surfaces, named $s_1, \ldots, s_5$; the constraint solver identifies a view as a subgraph of the whole model (circled in Figure 24.2). The variables in the subgraph could be associated with surfaces in the image; the others are labelled *invisible*. Object constraints involving *invisible* surfaces are considered satisfied; in a sense they are relaxed.

It is worth noticing that all the visibility information encoded in an Aspect Graph [74] can be encoded in Visual Constraints as well. In other words, the following property holds:

Property 4. *Every possible combination of visible and invisible surfaces in an aspect can be described with visual constraints.*

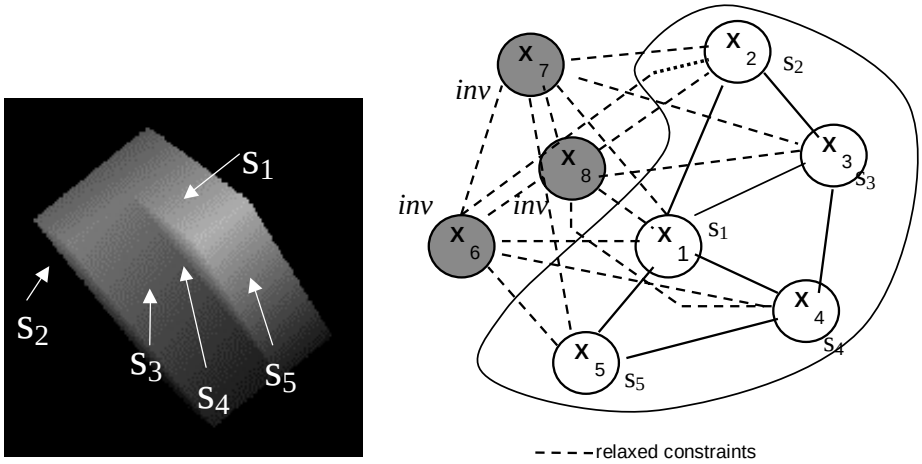

Figure 24.2 Model-Figure association

Proof. Given a polyhedron with N facets, let us consider the set A of possible aspects. The set A is a subset of $\{visible, invisible\}^N$ that provides the feasible combinations of visible/invisible surfaces. It is, thus, an N-ary visual constraint on the set of surfaces. □

Property 4 provides a method to obtain the set of visual constraints given the aspect graph: we can always compute a table of Boolean values providing visibility information in each possible view and consider this table as an N-ary visual constraint. In other words, given the aspect graph, we can compute the set V of possible views; if N is the set of visual features in the object model, we can provide a table $T(|V| \times N)$ such that $T(v, f) = true \leftrightarrow$ the feature f is visible in the view v.

Thus, it is possible to get the set of visual constraints in the many ways studied in the literature for the aspect graphs [30].

However, the set of visual constraints can be provided in more general ways. While the aspect graph is build from metrics information, or information about proportion of the model parts, the set of visual constraints can be constructed with more generic models. The set of visual constraints can be even defined without metrics information, allowing the recognition of wide classes of objects. For example, in order to recognize a parallelepiped, it is not necessary to know the lengths of the edges, nor their ratio. If the user wants to count the number of parallelepiped in an image, regardless of their dimensions, only the necessary constraints have to be given (e.g., perpendicularity and parallelism between

features), without any information about edge lengths. Of course, if we want a specific object and have more information, we can synthesize more selective visual constraints.

If the object is not very complex, the set of visual constraints can be defined by the user, using basic constraints like the ones in the examples in section 24.1. If the object is more complex, visual constraints can be inferred by the system, either exploiting the same techniques used for the aspect graph, or with a set of rules. As an example, consider the L-shaped object in Figure 24.1; in order to obtain a good recognition, more information can be exploited. For instance, human vision would not accept a solution that considers $X1$, $X3$, $X4$ and $X5$ as visible surfaces and $X2$, $X6$, $X7$ and $X8$ as invisible; in fact, if $X2$ is invisible then $X4$ cannot be visible. We impose a visual constraint `if_visible(X4,X2)` stating that whenever $X4$ is visible, $X2$ is visible as well. A possible way to impose this constraint is by means of rules that accept and provide information about constraints; a natural way can be Constraint Handling Rules [39, 40]. In our system, we imposed a set of Constraint Handling Rules like the following:

$$parallel(A, C), concave(A, B), touches(A, B), touches(A, C) ==> if\_visible([A, B], C).$$

stating that, if there are three surfaces A, B and C, A and C are parallel, A and B form a concave angle, A touches B and B touches C, then whenever you can see both A and B, C is visible.

It is worth noting that the concepts of *invisible* surface and *visual* constraints interface very easily with the ICSP. Visual constraints can be easily implemented in the interactive way, to acquire an invisible surface when necessary. For example, the visual constraint $hide(X_1, X_2)$ could acquire an *invisible* surface for the domain of X_2 if there is at least a visible surface in the domain of X_1.

25.0 Preferred solutions

In many cases, the set of visual constraints is not able to prune enough the search space, and some uninteresting solutions are returned with the interesting ones. As an example, consider a visual search system that looks for cubes in an image. A cube can appear with tree facets, with two or with one. This means that if a square facet is seen by the system, it will be possibly recognized as a cube. On the other hand, if the system can see three facets, it should not return as solutions the three assignments that consider visible only one of the three surfaces. In other words, suppose that the system is able to recognize in an image containing three facets (call them f_1, f_2 and f_3) a cube; a feasible assignment could be $\{X_1 \mapsto f_1, X_2 \mapsto f_2, X_3 \mapsto f_3, X_4 \mapsto invisible, X_5 \mapsto invisible, X_6 \mapsto invisible\}$. In this case, the assignment $\{X_1 \mapsto f_1, X_2 \mapsto invisible, X_3 \mapsto invisible, X_4 \mapsto invisible, X_5 \mapsto invisible, X_6 \mapsto invisible\}$ will be recognized, because there exists a possible view for a cube where only X_1 is visible.

As another example, consider the object in figure 25.1. The inner surface, S_{inside} is not visible from the current viewpoint, but would be visible if the slot was less deep. This situation cannot be expressed by means of visual constraints unless they consider more information, and this would make the model more complex.

Intuitively, if there is a "good" interpretation for some object O in the scene, we do not want the system to provide "worse solutions" that assign the same interpretation for a sub-part of the object O. At first glance the problem could be addressed by using a COP: we could add to the CSP a cost function that counts the number of invisible surfaces in each solution and select only the solutions that minimize the function. In this way, if two objects satisfying the model are present in the scene (viewed, eg., from different viewpoints), only one of the objects would be recognized. The system would be unable, for example, to count the number of objects that match with the model. For example, in Figure 25.2, we have two objects; both of them can be considered as an instance of the L-shaped object defined in Figure 24.1. However, if we employ the COP model, only the object on the bottom-left angle will be recognized, because it has five visible surfaces, instead of the four of the top-right object.

Another possibility would be defining a minimum number of visible surfaces i.e., with the cardinality operator [60], $\sharp([X \neq invisible, Y \neq invisible, Z \neq invisible]) \geq MinInvisible$. In the example of Figure 25.2, $MinInvisible$ should be four surfaces, in order to recognize

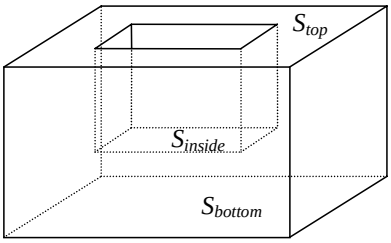

Figure 25.1 Example of 3D object

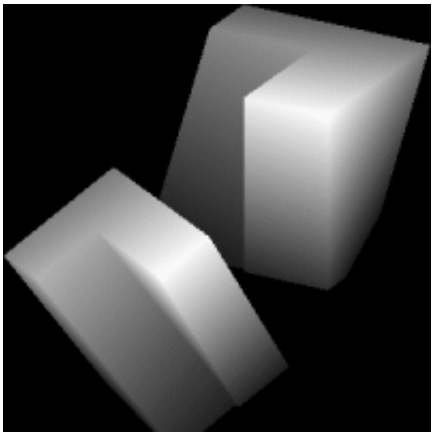

Figure 25.2 Example of 3D image

both the objects; but in this case, we would obtain six solutions referring to the object in the lower-left corner. In fact, if we substitute in an assignment a visible surface with a dummy value, we still have a feasible solution. This approach would make our system completely unable to address some problems, like counting the number of figures that match with the object model.

In our visual search problem, the set of assignments should not be totally ordered by means of an objective function, because different assignments are not always comparable. Some assignments refer to the same object in the image, and represent parts of the object. Others represent different objects and, thus, are not comparable. In other words, the feasible assignments should be ranked through a *partial order*: assignments referring to different objects are not comparable, while assignments that consider the same object but with a different number of *invisible* assignments are comparable.

This idea leads to the concept of Partially-ordered Constraint Optimization Problem, defined in the following section. Then, in section 26.3, we will define the partial order for

visual search.

26.0 Partially-ordered Constraint Optimization Problems

26.1 Definition

We now extend the concept of Constraint Optimization Problem to the case of a partial order among solutions [43]:

Definition 26.1. *A* Partially-ordered Constraint Optimization Problem *(PCOP) is a quadruple* $R = \langle P, f, G, \preceq \rangle$*, where* $P = \langle \{X_1, \ldots, X_n\}, \{D_1, \ldots, D_n\}, C \rangle$ *is a CSP,* $f : D_1 \times \cdots \times D_n \mapsto G$ *is a function,* G *is a set and* $\preceq$ *is a partial order on* G*. A* Solution *of the PCOP* R *is a solution* S *of* P *such that* $\nexists S'$ *solution of* P *and* $f(S) \prec f(S')$*.*

The set of PCOP solutions is a subset of the set of CSP solutions; more precisely, they are those solutions that are not dominated by other solutions.

26.2 Algorithms for solving PCOPs

A PCOP can be solved in a variety of ways, the most trivial (and inefficient) is finding all the solutions of the CSP and a posteriori select only the solutions of the PCOP. A more efficient approach is an extension of the Branch and Bound (B&B) procedure [78]. B&B is an efficient, widely used method for solving COPs; it could be described as follows: first we find a solution (typically using some kind of tree search), then we add a further constraint whose declarative semantics could be informally stated as "any new solution must be *better* than the current best". The operational semantics of the constraint can be given in different ways. In Constraint Programming it is usually intended as "*each new solution must have a greater merit value*". In Operations Research (and in the original B&B formulation) it is intended as "*the Upper Bound of each subproblem is greater than the merit value of the found solution*". Both the descriptions share the same declarative semantics for the added constraint, while provide different propagation and pruning powers. Of course this means keeping memory of the best solution achieved so far. Various implementations have been provided for this basic idea, for example in Constraint Logic Programming [89].

The method can be easily extended by considering, instead of a single additional constraint, a set of constraints that limit the next solutions to be better (in the $\preceq$ sense) than

the already achieved ones. In other words, to solve a PCOP, we have to store all the solutions of the CSP that are currently believed to be solutions of the PCOP as well (the set $CurSol$ in Figure 26.1).

```
15:let CSP = ⟨X, D, C⟩, PCOP = (CSP, f, G, ⪯)
16:CurSol = ∅                                   % Set of Current Solutions
17:While solutions can be found
18:            find a solution S of CSP
19:            CurSol ← (CurSol \ {x ∈ CurSol | f(x) ≺ f(S)}) ∪ {S}
20:            C ← C ∪ {not(f(X) ≺ f(S))} % Add a new constraint:
                                                % next solutions must be better than S
21:End While
22:Return CurSol
```

Figure 26.1 Sketch of B&B Algorithm for a PCOP

A more detailed description of the same algorithm is in figure 26.2, where the branching and bounding phases are clearer. The version in figure 26.1 redoes all the computation after finding a solution, while the one in figure 26.2 goes on in the tree search. In a sense, the version in figure 26.1 corresponds to implementation bb_1 in [89], or to `min_max` in ECL^iPS^e and CHIP [24] and to `IlcMinimize` in ILOG SOLVER [91].

Note that the propagation phase of $\preceq$ constraints is intentionally left undescribed, because it could be implemented in a variety of ways. For example, it could be implemented with or without bounds.

Another description of the same algorithm could be given in terms of two constraint stores: one is *local* and contains the constraints present in each node of the search tree; the other is *global* and its constraints are to be intended present in all the local constraint stores of all the search tree nodes. So, constraints in the global constraint store are managed as usual, with the usual transitions of CLP operational semantics [69]. In other words, constraints can be added to the Global Constraint Store, they can be removed if they are entailed by the store, or can cause failure, if the Store is inconsistent. In this way, at a certain computation step, we could have a failure of the global constraint store and stop the search, knowing that no better solutions can be found, thus pruning many branches of the search tree. Note that, in Figure 26.2 constraints are only added, because this is their

```
1: let CSP = ⟨X, D, C⟩, PCOP = (CSP, f, G, ⪯)
2: let AP = {CSP}                          % Set of Active Problems
3: CurSol = ∅                              % Set of Current Solutions
4: While AP ≠ ∅
5:            choose P ∈ AP, let P = ⟨X_p, D_p, C_p⟩
6:            if P is inconsistent
7:                  AP ← AP \ {P}
8:                  go to step 4
9:            if P has only one solution S_p
                    % Add the constraint not(f(X) ≺ f(S_p)) to all the active problems
10:                 ∀Q ∈ AP, Q = ⟨X_q, D_q, C_q⟩, C_q ← C_q ∪ {not(f(X_q) ≺ f(S_p))}
                    % remove from CurSol the solutions subsumed by S_P and add S_p
11:                 let RS = {S ∈ CurSol | f(S) ≺ f(S_p)}
12:                 let CurSol ← CurSol \ RS ∪ {S_p}
              Else  % Branch
13:                 Generate the set CP of children of P
14:                 AP ← AP \ P ∪ CP
15:End While
16:Return CurSol
```

Figure 26.2 B&B Algorithm for a PCOP

declarative behaviour; however they can be operationally removed from the constraint store whenever they are redundant (i.e., entailed by the other constraints of the store).

Often Branch-and-Bound is applied to a depth-first search with backtracking; since the added constraint is not retracted upon backtracking, it is also called *unbacktrackable constraint*. In the same way, in our extension we also call the global constraint store *unbacktrackable constraint store*.

In general, the PCOP deals with the problem of finding *all* the non-dominated solutions; and the PCOP-B&B algorithm is not a good way to tackle with problems in which only one solution is requested. In fact, if we have a CSP solution S_1, in order to show that S_1 is also a solution of the PCOP, we must show that no solution of the corresponding CSP can be better than S_1. For example, if one is interested in finding just one non-dominated solution, i.e., one CSP solutions for which we know that no better solution can be found, the search must still be exhaustive. In particular instances, however, it is possible to obtain

the nondominated solutions without exhaustive search, as will be explained in the following section.

26.3 PCOP for Visual Search

As stated informally in section 25, the set of solutions in visual search is naturally ranked in a partial order. In this section, we define formally a partial order and we show how it can be used in visual search problems.

We define the following ranking:

Definition 26.2. *Given two possible assignments* $A_i = \{X_1 \mapsto s_{i_1}, \ldots, X_n \mapsto s_{i_n}\}$ *and* $A_j = \{X_1 \mapsto s_{j_1}, \ldots, X_n \mapsto s_{j_n}\}$*, we say that*

$$A_i \preceq_v A_j \quad \Leftrightarrow \quad \forall k \in \{1, \ldots, n\},\ s_{i_k} = s_{j_k} \vee s_{i_k} = \mathit{invisible}$$

Since the space of possible assignments is already partially-ordered, we have a PCOP where the objective function f is simply the identity function:

Proposition 26.3.1. *Given a CSP* $P = \langle \{X_1, \ldots, X_n\}, \{D_1, \ldots, D_n\}, C \rangle$, $G \equiv D_1 \times \cdots \times D_n$ *and the identity function* I *on* G*, the quadruple* $R_v \equiv \langle P, I, G, \preceq_v \rangle$ *is a PCOP.*

Proposition 26.3.2. $\preceq_v$ *defines a semi-lattice on the set of possible assignments.*

Proof.

- $\preceq_v$ is a partial order, in fact it is reflexive ($A \preceq A$), antisymmetric ($A \preceq_v B \wedge B \preceq_v A \Rightarrow A \equiv B$) and transitive ($A \preceq_v B \wedge B \preceq_v C \Rightarrow A \preceq_v C$).
- There exists the *bottom* element of the semi-lattice: $\perp = \{\forall i, X_i \mapsto \mathit{invisible}\}$

□

Evidently, the top element does not exist in general, so $\preceq_v$ does not define a lattice. For example, given the two solution tuples $A_1 = \{X_1 \to s_1, X_2 \to \mathit{invisible}\}$ and $A_2 = \{X_1 \to s_2, X_2 \to \mathit{invisible}\}$, there is not an element E such that $A_1 \preceq_v E$ and $A_2 \preceq_v E$. The same problem could be described with a partial order that is also a lattice, because it is always possible to extend a partially-ordered set to a lattice; however, the obtained formalization is not always intuitive and useful.

In this particular case, however, we can avoid searching all the search space if we choose the heuristics that comes from the following observation:

Observation 26.3.1. *If we try to assign to a variable all the visible surfaces in its domain before the invisible one, then, after a solution is found, we know that no better solution exists.*

Proof. Suppose that we have a solution $S_i = \{X_1 \mapsto s_{i_1}, \ldots, X_n \mapsto s_{i_n}\}$ of the CSP and a better solution $S_j = \{X_1 \mapsto s_{j_1}, \ldots, X_n \mapsto s_{j_n}\}$ exists (i.e., $S_i \preceq_v S_j$). Let us suppose, for ease of presentation, that the variables are labelled in lexicographic order (i.e., $X_1, X_2, \ldots, X_n$). From definition 26.2, there must be an index k such that $s_{i_k} = \mathit{invisible}$ and $s_{j_k} \neq \mathit{invisible}$ and $\forall l \neq k,\ s_{i_l} = s_{j_l}$. But, if the assignment $X_k \leftarrow s_{j_k}$ is consistent with constraints w.r.t variables $X_1, \ldots, X_{k-1}$, then it was tried before the assignment $X_k \leftarrow \mathit{invisible}$ and solution S_j was found before S_i. If the assignment $X_k \leftarrow s_{j_k}$ is inconsistent with constraints, then S_j is not a solution. □

Thus, if we find a (CSP) solution exploiting this heuristics, we know that it is also a solution of the PCOP and we do not need the general B&B algorithm shown in section 26.1. In section 26.2 we hinted that for general PCOP problems, exhaustive search is necessary in order to show that an assignment is a non-dominated solution. In this visual search problem, however, it is not necessary, and recognition of an object can be performed without looking for all the possible solutions. In other words, the use of a PCOP algorithm for visual search does not invalidate the efficiency result of ICSP, since we are not forced to acquire the whole set of visual features to find a solution.

In general, however, PCOPs cannot be solved this simply; other interesting instances can be solved with the Branch-and-Bound extension given in section 26.2. In the next section, we show one of these instances and compare our algorithm with other techniques.

26.4 Pareto Optimality as a PCOP

In the Pareto optimality problem, the scope is dealing with multi-objective optimization. For our needs, we will define the Multi-objective Optimization Problem as follows:

Definition 26.3. *A* Multi-objective Optimization Problem *(MOP) is a quadruple* $R_M = \langle P, \overline{f} = (f_1, \ldots, f_m), T, \leq \rangle$ *such that* P *is a CSP and* $f_1, \ldots, f_m$ *are functions* $f_i : D_1 \times \cdots \times D_n \mapsto T$ *and* $\langle T, \leq \rangle$ *is a totally ordered set.*

We will suppose to deal with a multiple maximization problem (i.e., we want to maximize all the f_i functions), being straightforward the extension for dealing with minimization of some of the functions.

Definition 26.4. *Given $A, B \in T^m$, we say that B* dominates *A, written $A \preceq_p B$ iff $\forall_{i=1}^m A_i \leq B_i$.*

Definition 26.5. *Given a MOP $R_M = \langle P, \overline{f}, T, \leq \rangle$, an assignment S is a* non-dominated solution *if it is a solution of P and $\nexists S'$ solution of P and $S \preceq_p S'$.*

We will consider as solution of the MOP the set of non-dominated CSP solutions. In order to define the MOP as an instance of PCOP, we need to define (see Definition 26.1): (i) the CSP P, (ii) the set G, (iii) the partial order $\preceq$, (iv) the function f which maps solutions to S_p. We define f as the function $\overline{f}$ of the MOP, that maps solutions to the set T^m (also called *the criterion space* [109]). The set T^m is sorted with the partial order:

$\preceq_p$ is a partial ordering on the set of possible results T^m, moreover, it is a lattice.

We can now formally cast the MOP as an instance of the PCOP framework:

Proposition 26.4.1. *Given the MOP $R_M = \langle P, \overline{f} = (f_1, \ldots, f_m), T, \leq \rangle$ and the partial order $\preceq_p$ as defined in definition 26.4, the quadruple $R_p \equiv \langle P, \overline{f}, T^m, \preceq_p \rangle$ is a PCOP.*

The constraints added in the Branch & Bound procedure, will be thus $not(\overline{f(X)} \prec_p \overline{f(S)})$; in other words, a (tentative) possible solution X will be pruned off if it is dominated by an already obtained solution S, i.e., if S is better in all the objective functions: $\forall_{i=1}^m f_i(X) \leq f_i(S)$. Of course, this is the declarative meaning of the constraint; operationally, it could be implemented in wiser ways; e.g., if we have some upper bounds U_i on the objective functions, we could state: $\forall_{i=1}^m U_i(X) \leq f_i(S)$.

These constraints could be very memory-consuming, depending both on the granularity of the criterion space and of the solution space. In the worst case, whenever we add a constraint, no other constraint is eliminated (because no previous element is subsumed). The number of stored constraints is bounded both by the number of CSP solutions (that is $\leq |D|^n$) and by the size of the criterion space (T^m). The algorithm provides a lot of information (the whole non-dominated frontier) and it is complete, so it can be, not surprisingly, expensive.

The presented partial order, $\preceq_p$, is not the only possible order useful in a Pareto Optimality problem. In some cases, the user is not interested in the whole non-dominated

frontier, but just in a subset of the frontier, or in a set of solutions that are *"near enough"* to the Pareto front.

However, the algorithm can work with any partial order, so we could consider a different one. Intuitively, we could draw the non-dominated frontier at "low resolution" by reducing the granularity of the criterion space. In other words, we could split the range $T = [L..U]$ into segments each of size r and consider the new space $T' = \{[L..L + r],]L + r, L + 2r], \ldots,]U - r, U]\}$. Now the new (low-resolution) problem would be

$$R'_M = \langle P, (f'_1, \ldots, f'_m)\rangle,$$

where

$$f'_i : D^n \mapsto T', X \mapsto I, s.t. I \in T' \wedge f_i(X) \in I.$$

Solving problem R'_M is less memory consuming, since the number of stored constraints cannot exceed the cardinality of the criterion space $(T')^m$. The information obtained by solving problem R'_M is the following: in each "pixel" of the graph in criterion space there is at least a solution of the original problem R_M; the viceversa is not always true, as shown in Fig. 26.3. Then, the user can be provided the "low resolution" graph and he/she will point the most interesting square. Now, we can refine the problem; supposing that the user selected the hypercube $H = \{\overline{V} | v_i \in (l_i..u_i)\}$, the refined multi-objective problem is $R"_M = \langle P', \overline{f}\rangle$, where $P' = \langle X, D, C \cup \overline{f} \in H\rangle$; i.e., a constraint is added to the original formulation P_M stating that the solution should fall in the hypercube. The new problem $R"_M$ is simpler than R_M, because of three reasons. First its space complexity is bounded by the size of the hypercube: r^m. Second, we have both a lower bound for the desired solution and an upper bound (for each criterion f_i); this can help in the bounding phase: both the bounds can perform pruning. Third, we already know a solution in the hypercube, so we already have a good lower bound (in the $\preceq$ sense).

This "zooming" can be extended to an arbitrary number of levels, so an arbitrary accuracy can be obtained. In a sense, this can be considered an *interactive* method [109].

It is worth noting that with the described "zooming" method, not all the nondominated points are achieved. For instance, in Figure 26.3 some points are shown that do not belong to any gray square. If we want to obtain the whole nondominated frontier, we do not have to add the lower bound; i.e., the refined MOP $R"_M$ should not contain the constraint $\overline{f} \in H$ but $\forall i, f_i \leq u_i$.

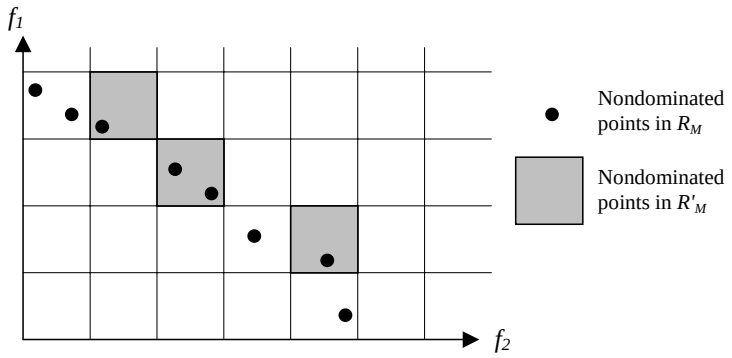

Figure 26.3 Criterion Space of a MOP in high and low resolution

In the following sections, we compare our algorithm with the usually used algorithm for the Pareto frontier. We first compare from a theoretical viewpoint, then we give experimental results.

26.4.1 Theoretical Comparison

26.4.1.1 Algorithms van Wassenhove - Gelders and Iterative B&B.

Usually, the problem of finding the non-dominated frontier is addressed by finding the optimal value in one direction and restart the search with constraints that limit the search space to other Pareto-Optimal solutions. This algorithm has been widely used [51]; the first reference we are aware of is by Van Wassenhove and Gelders [117]. They used the idea to find the nondominated frontier in a bicriterion scheduling problem. The algorithm can be described as in Figure 26.4.

1. find the optimal solution wrt. function f_2; let us call it O_2. Let $\Delta \leftarrow O_2$
2. impose the constraint $f_1 > f_1(\Delta)$
3. minimize f_2. If a solution π^* exists, then it is efficient; else go to Step 5.
4. Let $\Delta \leftarrow f_1(\pi^*)$. Go to Step 2.
5. Stop.

Figure 26.4 Algorithm Van Wassenhove and Gelders

Due to the complexity of the proof, we will not compare our algorithm (PCOP-B&B) with Wassenhove-Gelders (WG). We will compare with an algorithm that can be considered as a limit of (WG) when the number of non-dominated solutions is very high. It is worth noting that (as we will see in the computational results in section 26.4.3) if the set of nondominated solutions is very low, the problem is probably much more simple; so in this theoretical approach we are addressing difficult problems. We call this algorithm *Iterative Branch-and-bound* (Figure 26.5).

1. find the optimal solution wrt. function f_2; let us call it O_2
2. build b problems: $P_1, \ldots P_b$. In each P_i, add the constraint $f_1 \leq f_1(O_2)\frac{i}{b}$ (thus $P_b \equiv P$).
3. find the optimal solution of $P_1 \ldots P_{b-1}$ wrt function f_2

Figure 26.5 Algorithm Iterative Branch & Bound

In algorithm 26.5 we divide the criterion space in b strips corresponding to non intersecting intervals of function f_1 and we optimize them with respect to function f_2 (Fig. 26.6). b optimization steps are performed: one to calculate O_2, the optimum of function f_2 (that in the first iteration coincides with Δ) and $b-1$ to calculate the rest of the frontier. Note that this algorithm is able to provide the whole non-dominated frontier only if b is big enough. Note also that the algorithm does not compare solutions with respect to the concept of non-domination. In fact, considering function f_1, only the strips are compared. In other

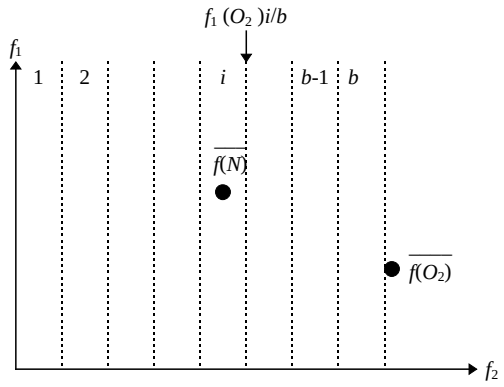

Figure 26.6 Division of criterion space in algorithm Algorithm Iterative B&B

words, it considers a solution A dominated by another B iff B belongs to a same or better strip of A for what concerns the function f_1 and B is better than A for what concerns function f_2. Formally,

$$\overline{f(B)} \preceq_s \overline{f(A)} \Leftrightarrow f_2(B) \leq f_2(A) \wedge \left\lceil \frac{f_1(B)}{f_1(O_2)} b \right\rceil \leq \left\lceil \frac{f_1(A)}{f_1(O_2)} b \right\rceil \tag{26-1}$$

Note that, if the number of non-dominated solutions is very big (like in very difficult problems), there will probably be a non-dominated solution in every strip; in this case algorithm 26.4 explores the same nodes as algorithm 26.5 when using the same partial order (equation 26-1).

We will now compare with the algorithm PCOP-B&B, given in Figure 26.2. Note that this algorithm depends on a partial order: by defining the partial order $\preceq$ we can obtain different sets of solutions. If we define $\preceq$ as the partial order that defines the concept of domination in a MOP, (definition 26.4), then we obtain the whole non-dominated frontier. If we use the same partial order of equation 26-1, we will obtain the same solutions provided by algorithm 26.5 (or equivalent solutions, where $A \equiv B \Leftrightarrow A \preceq B \wedge B \preceq A$. We obtain the same solutions if we expand the nodes of the search tree in the same order).

26.4.1.2 Comparison between PCOP-B&B and Iterative B&B.

Let us suppose that the two algorithms use the same strategy for searching the search space, i.e., the nodes of the search tree are expanded in the same order: $\ll$. For the sake of comparison, we need to have the same coarseness for the problem for both algorithms. Thus, we need to have the same partial order, we will consider for both algorithms the ordering given in equation 26-1.

Note that, in order to do a fair comparison, we need to know the partial order in advance. Since the partial order is defined in terms of O_2 (see algorithm 26.5), we need to consider in both algorithms a first step of optimization. In other words, we add step 1 of algorithm 26.5 as first step of algorithm 26.1 (see Figure 26.7).

Note that the condition in line 3 restricts the search to the first $b-1$ strips (Prop. A.0.1): it is fair, in fact also Algorithm 26.5 searches only the first $b-1$ strips, after the first optimization step.

The following observation will be useful in successive proofs:

1. find the optimal solution wrt. function f_2; let us call it O_2
2. Consider the partial order among possible assignments $\preceq_s$ (eq. 26-1)
3. add the constraint $\neg\left(\overline{f(O_2)} \preceq_s \overline{f(X)}\right)$
4. restart the search; each time you find a solution S, impose the unbacktrackable constraint $\neg\left(\overline{f(S)} \preceq_s \overline{f(X)}\right)$

Figure 26.7 PCOP Branch & Bound with order $\preceq_s$

In order to prove that a node N is not explored by algorithm 26.7, we need to find a better node *that was explored before* N. The following observation tells that we do not need an *explored* node, but we just need a better node that is before N:

Observation 26.4.1. *In algorithm 26.1 (and in 26.7 as well) a node N is not explored if there is a feasible solution B such that $\overline{f(B)} \preceq \overline{f(N)}$ and $B \ll N$.*

Proof. If B is an explored node, the proof is trivial.

If B was not explored and it is feasible, it means that there is a solution B' that was explored before B and $\overline{f(B')} \preceq \overline{f(B)}$, thus, from the transitivity of $\preceq$, $\overline{f(B)} \preceq \overline{f(N)}$, so N is not explored. □

Lemma 26.4.2. *If algorithm Iterative B&B (Figure 26.5) does not visit a node in any of the problems in $P_1, \ldots, P_{b-1}$, then algorithm 26.7 does not visit node N as well, provided the same order of valuation of nodes $\ll$.*

Proof. Let us suppose that algorithm 26.5 does not visit a branch N in any of the subproblems $P_1, \ldots, P_{b-1}$. Let j be the index of the strip where $\overline{f(N)}$ lies:

$$j = \left\lceil \frac{f_1(N)}{f_1(O_2)} b \right\rceil \tag{26-2}$$

Since node N is pruned in P_j,

$$\exists B_j \ll N \text{ s.t. } f_2(B_j) \leq f_2(N) \wedge f_1(b_j) \leq f_1(O_2)\frac{i}{b} \tag{26-3}$$

A node N is pruned in algorithm 26.1 if

$$\exists B \ll N \ s.t. \ \left(\left\lceil \frac{f_1(B)}{f_1(O_2)} b \right\rceil \leq \left\lceil \frac{f_1(N)}{f_1(O_2)} b \right\rceil \right) \wedge f_2(B) \leq f_2(N) \tag{26-4}$$

If we take $B = B_j$, from eq. 26-3 we have that $f_2(B_j) \leq f_2(N)$ and

$$f_1(B_j) \leq f_1(O_2) \frac{j}{b}$$

thus

$$\frac{f_1(B_j)}{f_1(O_2)} b \leq j$$

Since j is integer, we also have that

$$\left\lceil \frac{f_1(B_j)}{f_1(O_2)} b \right\rceil \leq j$$

and, from the definition of j (eq. 26-2)

$$\left\lceil \frac{f_1(B_j)}{f_1(O_2)} b \right\rceil \leq \left\lceil \frac{f_1(N)}{f_1(O_2)} b \right\rceil$$

□

So we can say that:

Theorem 26.4.3. *Algorithm 26.7 visits less nodes than algorithm 26.5*

Proof.

- Nodes that are not visited in any problems from $P_1, \ldots, P_{b-1}$ are not visited by Algorithm 26.7
- Nodes that are visited in only one of the subproblems $P_1, \ldots, P_{b-1}$ are visited once.
- Nodes that are visited in more than one of the subproblems $P_1, \ldots, P_{b-1}$ are visited once.

□

26.4.2 Specializing PCOP Branch-and-Bound for Pareto Optimality

The generic algorithm described in Figures 26.1 and 26.2 can be specialized for the different types of problems falling in the PCOP framework. In the case of Pareto Optimality, some observations can lead us to a more efficient implementation of the unbacktrackable constraint store.

In many CLP(FD) systems [2, 14] each objective function can be represented as a domain variable, i.e., a variable with a domain. Considering $\overline{f} = (f_1, f_2, \ldots, f_m)$ the vector of all the objective functions, the domain of the vector function $\overline{f}$ is the cartesian product of the domains of the scalar functions $D_{\overline{f}} = D_{f_1} \times D_{f_2} \times \cdots \times D_{f_m}$. Ideally, we would like to delete from the domain $D_{\overline{f}}$ all the area forbidden by the unbacktrackable store; but we can only restrict the domains of the functions f_i, and $D_{\overline{f}}$ must always be the Cartesian product of the scalar domains. The following property provides a method to decide if a domain can be reduced by the unbacktrackable store.

Property 5. *Let S be the set of previously found solutions, $\overline{f} = (f_1, f_2, \ldots, f_m)$ the objective function, $\forall_i D_{f_i}$ is the domain of f_i, $LUB_i = max(D_{f_i})$, $GLB_i = min(D_{f_i})$. Let us call $DP_i \equiv (LUB_1, LUB_2, \ldots, LUB_{i-1}, GLB_i, LUB_{i+1}, \ldots, LUB_m)$; i.e., the vector which is "best in all possible directions but direction i".*

The domain $D_{\overline{f}}$ can be reduced by the unbacktrackable store only if $\exists i$, $\exists s \in S$ such that $DP_i \prec_p \overline{f(s)}$

Proof. Suppose that the value $v_i \in D_{f_i}$ can be deleted from D_{f_i}. This means that all the values of $\overline{f}$ for which $f_i = v_i$ are dominated by some previously found solution.

In particular, there will be a solution s that dominates the point $(LUB_1, LUB_2, \ldots, LUB_{i-1}, v_i, LUB_{i+1}, \ldots, LUB_m)$ so, for transitivity, $DP_i \prec_p \overline{f(s)}$. □

Property 5 explains that in order to perform a domain reduction, we only need to check if m points lie in the forbidden area. For example, consider Figure 26.8. The dotted area is forbidden by the unbacktrackable store, represented as a set of non-dominated solutions. The domain $D_{\overline{f}}$ is the rectangle in the middle, as it is the cartesian product of the domains D_{f_1} and D_{f_2}. Since we only consider reductions of the domains D_{f_1} and D_{f_2}, we can only delete the hatched area. This means that a reduction of $D_{\overline{f}}$ can occur only if the points DP_1 and DP_2 (indicated by the arrows) lie in the forbidden (dotted) area.

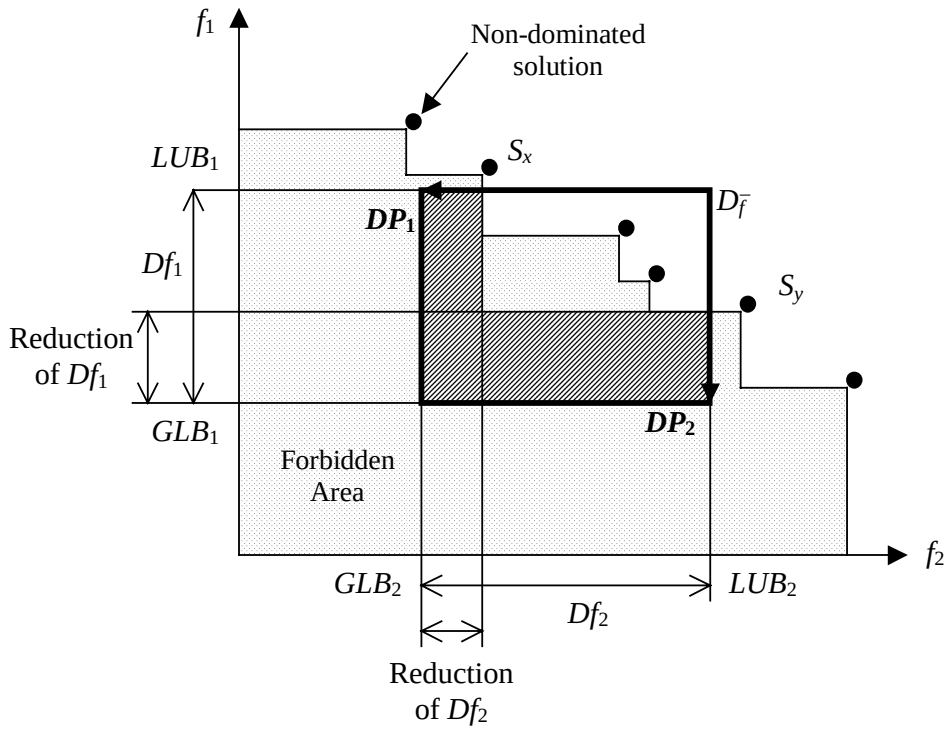

Figure 26.8 Region forbidden by the unbacktrackable constraint store

From this observation follows that we do not have to check the current domain of $\overline{f}$ against all the previously found solutions, but only against at most m previously found solutions (the solutions labelled S_x and S_y in Figure 26.8).

Of course, we need to arrange the previously found solutions in a suitable data structure, in order to avoid many useless constraint checks. Since the previously found solutions are points in a space, using spatial data structures seems wise.

In our experiments, we adopted the Point Quad-Tree representation [98], because it is very suitable for the Pareto Optimality problem. A Point Quad-Tree (in the following, simply Quad-Tree) is a tree that contains points in a space; historically, it has been defined for two dimensions [31], but can be easily extended to any number of dimensions. For ease of presentation, we will give our examples in the two dimensional case, but they are expandable to the N-dimensional case unless stated otherwise. A Quad-Tree can be considered an adaptation of the binary search tree to two dimensions and it is based on the principle of recursive decomposition of the original space. Each element of the tree is either a leaf or a branch node. In two dimensions, each branch node is identified by a couple of coordinates

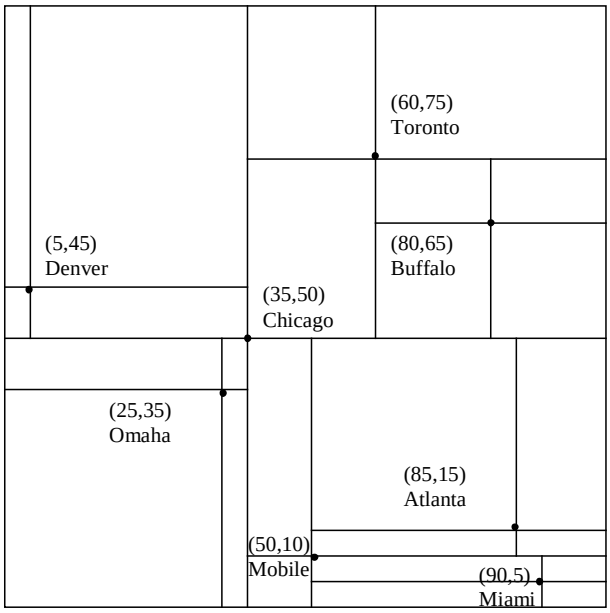

Figure 26.9 Example: records in a Quad-Tree

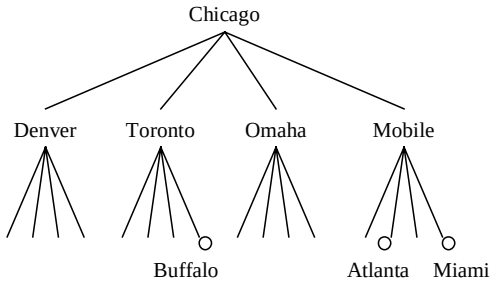

Figure 26.10 Example: a Quad-Tree

(X, Y) and has four children, representing the four quadrants in which the space is divided by the two cartesian axes translated into the point (X, Y). The quadrants are normally labelled NW (for North-West), NE, SW and SE. In Figure 26.10 we can see an example of a Quad-Tree and in Figure 26.9 the records it represents.

We believe that the Quad-Tree representation of the set of (currently believed) non-dominated solutions is suitable for a series of reasons. (i) First of all, the Quad-Tree allows to access the data structure in $O(log|S|)$ time complexity, if S is the current set of non-dominated solutions. This feature is not peculiar of the Quad-Tree; other possible spatial data structures, like K-d trees and PK Trees show the same feature [99]. (ii) The *domination* between points is easily verifiable: the area dominated by a point in a Quad-Tree is simply

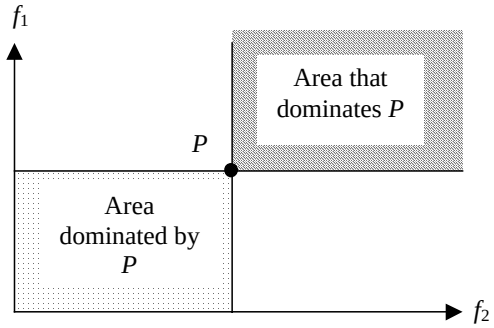

Figure 26.11 Domination in a Quad-Tree

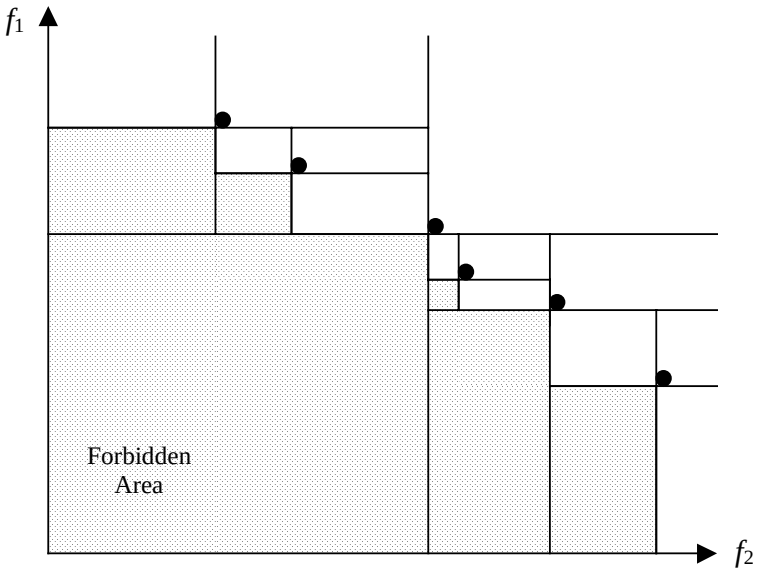

Figure 26.12 A Quad-Tree correspondent to a set of non-dominated solutions

the SW quadrant, while the area that dominates a point is the NE quadrant (Figure 26.11). (iii) The insertion of points can be obtained in $O(log|S|)$ time complexity.

As an example, in Figure 26.12 we can see a possible Quad-Tree representation of the set of solutions in Figure 26.8. The Quad-Tree corresponding to a set of points is not unique: it depends on the insertion order.

One drawback of quad-trees is that the deletion of a point can be an expensive operation. The algorithm by Samet [97] is often used; it consists of (i) finding a suitable new candidate and (ii) re-inserting some of the children of the node.

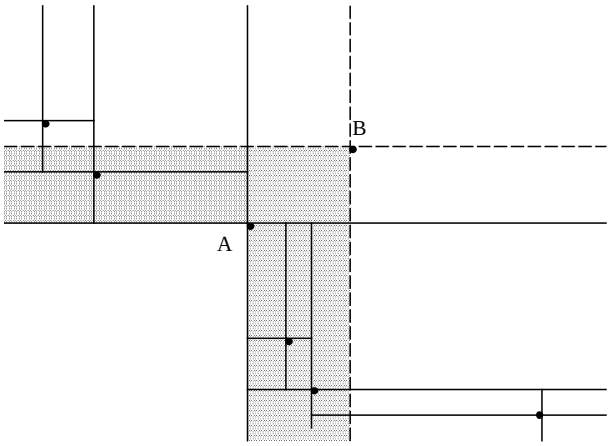

Figure 26.13 Point deletion in a Quad-Tree storing information for multi-objective optimization

However, in our instance, a deletion occurs only if a better point is found, thus, a possible strategy is to substitute the old point with the new one. In this case, many of the points that should be re-arranged in the Samet algorithm [97] are dominated by the new solution, so they can be deleted as well. In Figure 26.13, we can see that if the point A is deleted, then some other point dominates it (suppose point B). In this case, we can decide that the new root is B; we do not need to re-arrange all the points in the gray area, because they are dominated by B as well.

Another strategy could be to simply ignore the elimination of points: remember that unbacktrackable constraints are only added to the store; operationally they can be deleted when they are redundant, but it is not strictly necessary.

26.4.3 Experimental Results

We performed some experiments with a multi-knapsack problem and with randomly-generated problems.

A knapsack problem consists of a set of items, with weight and profit associated to each item, and a total capacity of the knapsack. The task is to find a subset of items that fit into the knapsack and maximizes the profit [82]. The single-objective problem can be extended to the multi-objective case by allowing an arbitrary number of knapsacks [120]: given a set of o items and a set of m knapsacks, with

$p_{i,j}$ = profit of item j according to knapsack i,

Number of items	Wassenhove-Gelders	PCOP-B&B	PCOP+Quad-tree
17	3.15	1.59	1.39
18	8.58	3.59	2.77
19	7.74	4.88	4.38
20	20.52	10.51	7.74
21	46.51	23.14	20.31
22	59.61	39.21	29.2
23	185.52	73.08	57.75
24	178.35	84.62	73.945
25	229.5	106.63	94.41
26	250	147.86	133.98
27	600.22	248.85	196.19
28	888.8	392.81	325.12
29	2133.42	730.3	659.23
30	3693.23	1305.33	1207.17

Table 26.1 Experimental results on Multi-Knapsack with 2 objective functions

$w_{i,j}$ = weight of item j according to knapsack i,

c_i = capacity of knapsack i,

find a vector $\overline{x} = (x_1, x_2, \ldots, x_o) \in \{0,1\}^o$, such that

$$\forall i \in \{1, 2, \ldots, n\} : \sum_{j=1}^{o} w_{i,j} x_j \leq c_i$$

and for which $\overline{f(x)} = (f_1(\overline{x}), f_2(\overline{x}), \ldots, f_m(\overline{x}))$ is maximum, where

$$f_i(\overline{x}) = \sum_{j=1}^{o} p_{i,j} x_j$$

and $x_j = 1$ iff item j is selected.

We compared our algorithm with the algorithm by Van Wassenhove and Gelders (Figure 26.4); we can see the results we had on Table 26.1 (results obtained with ECL^{i}PS^e [2] running on a with a 4 x UltraSPARC II 300 MHz). We can see that the PCOP Branch-and-Bound performs better and the Quad-Tree optimization gives a further improvement. Our method is applicable to problems with more than two objective functions; in Table 26.2 some timing results are given for problems with more criteria (results obtained with ECL^{i}PS^e running on a Pentium II 400 MHz with Linux).

We also performed some tests on randomly generated CSPs. The problems were generated in the same ways as for the tests in the ICSP (see Section 17). For simplicity, we

Number of Items	2D	3D	4D
10	0.04	0.09	0.14
11	0.06	0.11	0.33
12	0.09	0.2	0.46
13	0.13	0.33	0.97
14	0.21	0.68	1.76
15	0.34	1.62	3.45
16	0.4	2.25	9.11
17	0.79	5.67	10.31
18	1.8	7.9	24.72
19	2.45	16.98	53.47
20	4.7	24.06	144.97
21	11.37	57.04	286.65
22	18.07	63.31	507.9

Table 26.2 Experimental results on Multi-Knapsack with more objective functions; algorithm PCOP-B&B with Quad-Tree optimization

randomly generated linear objective functions. In the graph, each bar represents the average of ten problems, with number of variables = 10, size of domains = 15, constraint density ranging from 20 to 100%, constraint tightness from 10 to 90%. As we can see from the charts in Figures 26.14 and 26.15 the PCOP-B&B algorithm performs better in the difficult instances (note that the scale is logarithmic), while keeping the original behaviour in the easy instances.

In Figure 26.16 the ratio of the timing results for the same problems is shown. We can easily see that PCOP Branch-and-Bound is up to 18 times faster than the algorithm by van Wassenhove-Gelders, while it is never worse than 32%. It is worth noting that the instances in which the ratio is low (i.e., our algorithm is slower than Wassenhove-Gelders) are usually easy instances. For example, the worst ratio is 0.75, which is due to problems solved on average in 0.082 seconds by PCOP-B&B and in 0.062 by Wassenhove-Gelders. On the other hand, one of the best ratios (nearly 18), was obtained in a problem where van Wassenhove-Gelders took 13067 seconds (about 3 hours and a half) while our method was able to find the non-dominated frontier in about 12 minutes.

From our experiments, we understood that a Pareto optimality problem is complex if the Pareto set is big. If the problem is unsolvable, or it has one solution, it has the same complexity of the corresponding CSP. If it has only one non-dominated solution, it is equivalent to the corresponding COP: there is only one solution that is optimal for all

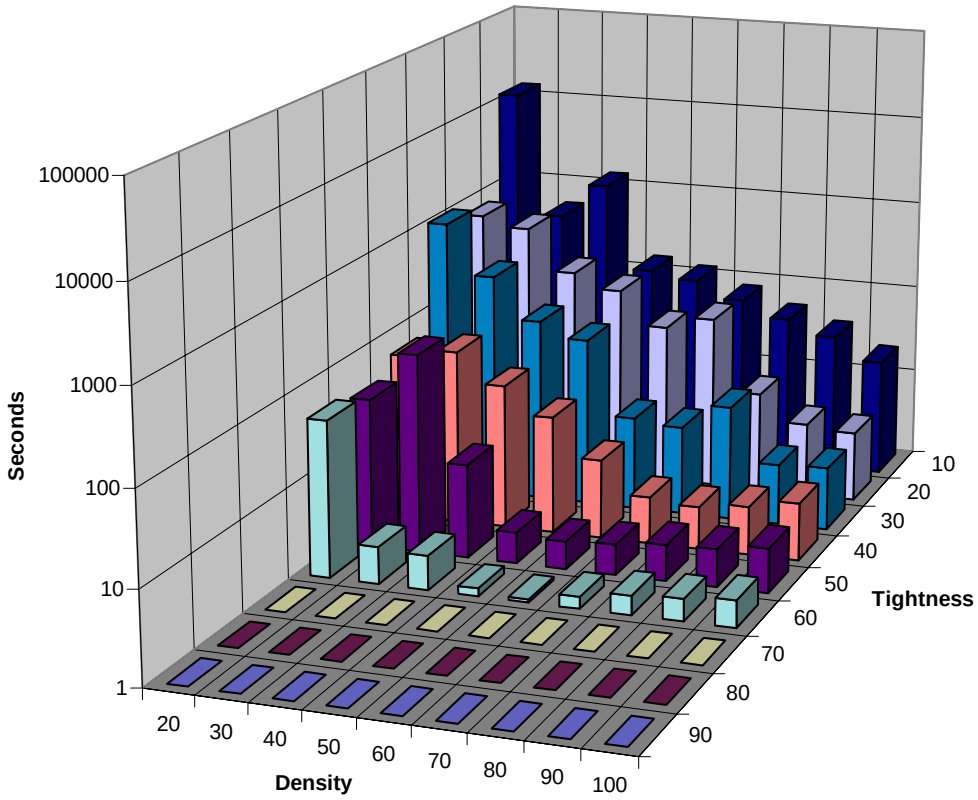

Figure 26.14 Performance of Wassehove-Gelders algorithm on randomly-generated MOPs

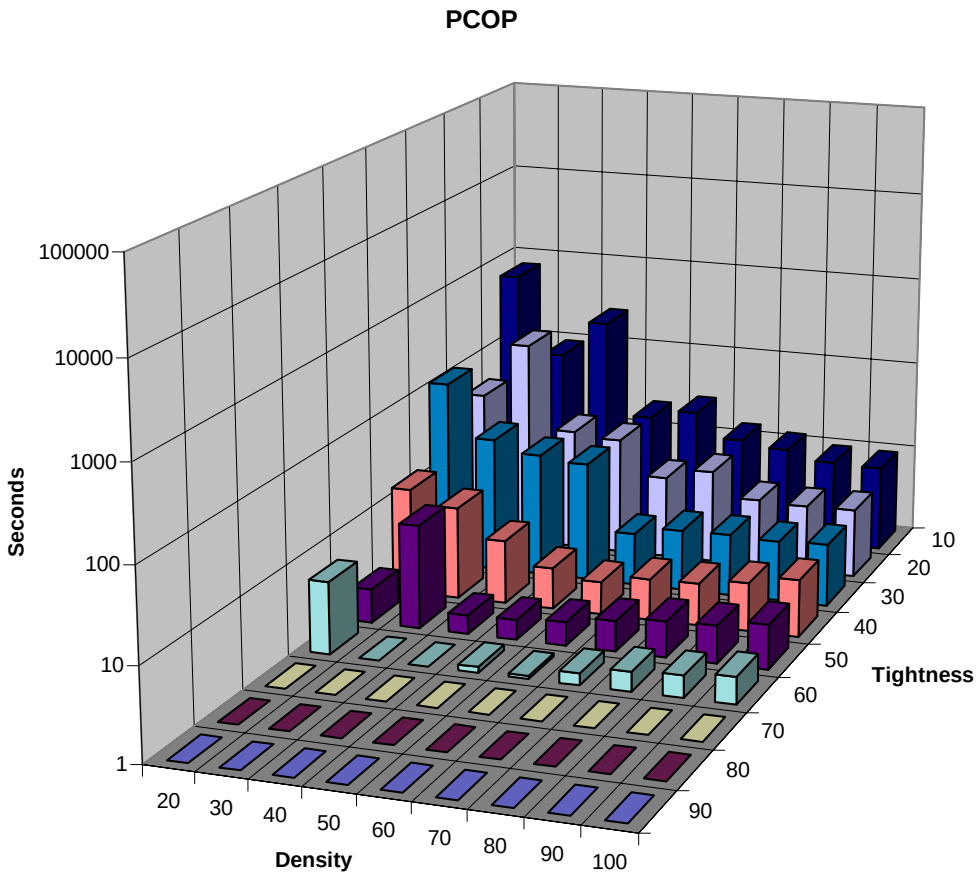

Figure 26.15 Performance of PCOP-B&B on randomly-generated MOPs

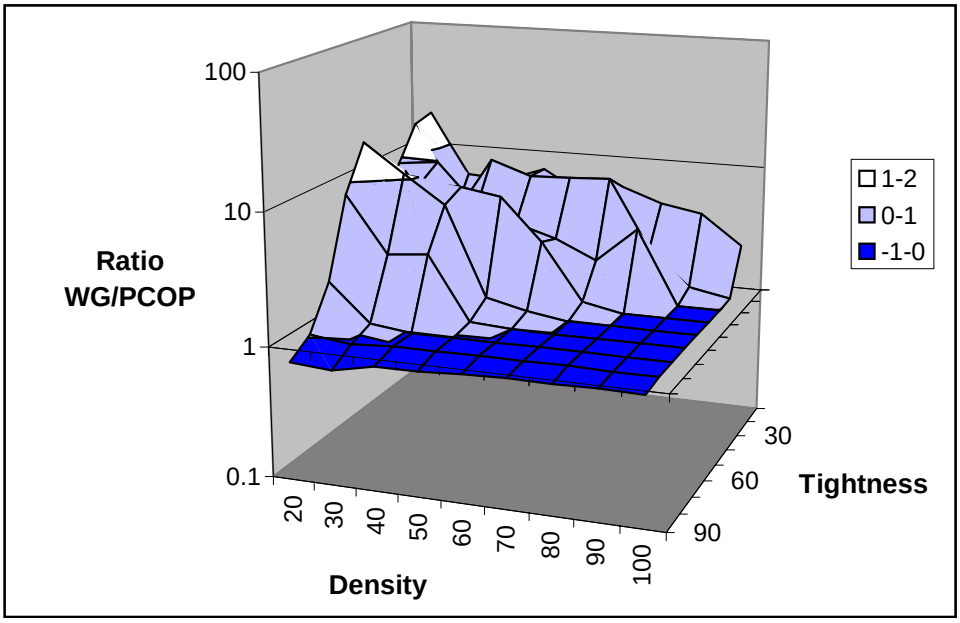

Figure 26.16 Ratio of computation time of WG and PCOP

the objective functions. If the number of non-dominated solutions is higher, then the time required to find the Pareto set can be high.

For this reason, we represented the ratio of the timing results as a function of the number of non-dominated solutions. From the graph in Figure 26.17 we can do the following observations: (*i*) again, in most cases algorithm PCOP Branch-and-Bound is faster than Van Wassenhove and Gelders (in fact, most of the dots are over the line $ratio = 1$). (*ii*) Most of the points are below the line $ratio = |PS|$, where PS is the Pareto set. (*iii*) The points are evenly distributed in vertical lines under the line $ratio = |PS|$.

One possible explanation is the following: as a lower bound, algorithm PCOP Branch-and-Bound will take the same time as Van Wassenhove and Gelders. As an upper bound, it can run up to $|PS|$ times faster than Van Wassenhove and Gelders; in fact, in order to obtain the non-dominated frontier, (WG) must restart the search at least $|PS|$ times.

26.4.4 Related works in Multi-Objective Optimization

Usually, these problems are addressed by translating the problem into a single objective problem (or a set of single objective problems) and then solved through standard single objective algorithms. Some methods convert the MOP into a COP. The *Weighting Objective*

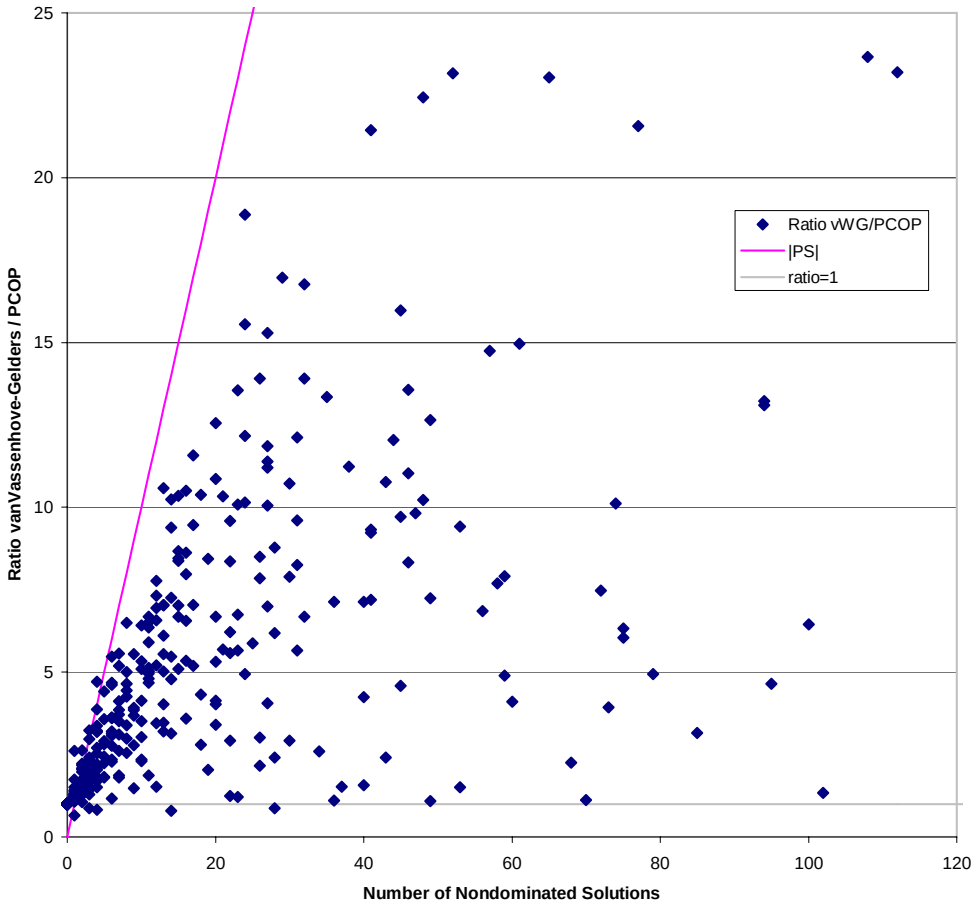

Figure 26.17 Ratio of computation time of WG and PCOP as a function of the number of non-dominated solutions

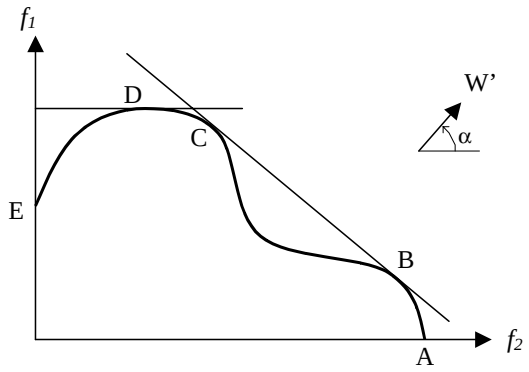

Figure 26.18 Example of concave non-dominated frontier

Method [109] considers a linear combination of the objective functions, with positive coefficients that sum up to one. The objective function is then given by $f = \sum_{i=1}^{m} w_i f_i$, for some w_i such that $\sum_i w_i = 1$ and $\forall i, w_i \geq 0$. Since there is only one objective function, standard methods can be applied. The coefficients are assumed beforehand, and they can be varied to obtain other solutions. However, if the problem's criterion space (i.e., the image through the objective functions of the set of solutions) is not convex, there is no guarantee that all the nondominated points can be generated. For instance, see figure 26.18, where the criterion space of a problem is represented. The feasible set is delimited by the curve and the two axis; we want to maximize both f_1 and f_2. The set of nondominated points is the part of the curve between points A and D, but the part comprised between B and C cannot be obtained with this method. In fact, maximizing the function f means maximizing in the direction described by vector $W = (w_1, \ldots, w_m)$, i.e., finding the point in the feasible criterion space that touches a line perpendicular to W. For vectors whose angle is greater than α the best points are in the arc $\widehat{CD}$, while for angles less than α the best solutions are in the arc $\widehat{AB}$. Sometimes [54] the part of the frontier generated through weighting vectors is called the set of *supported points*.

The *Hierarchical Method* and *Trade-Off Method* rank the objective functions in order of importance. A solution S' is found optimizing only the most important function (say f_1), then function f_1 is converted into a constraint; the obtained problem is then solved optimizing with respect to function f_2 and so on.

The *Global Criterion Method* [96] tries to minimize some kind of distance from the ideal solution. The ideal solution is computed by solving all the COPs with only one objective function. The ideal value for the function f_i is obtained by solving the problem with only f_i

as objective function; in this way an ideal vector $S^0 = (s_1^0, \ldots, s_m^0)$ is obtained. Now a new COP is solved whose objective function is $f(x) = \sum_{i=1}^{m} \left(\frac{s_i^0 - f_i(x)}{s_i^0} \right)^p$, where p is a parameter defining the kind of employed distance.

In *Goal Programming* [109] [66] the objectives are conceptualized as goals, then priorities or weights are defined for goals. Now, deviational variables are defined to measure how much a given goal has been achieved. For instance, for an objective $max(f_i(x))$ (i.e., maximize $f_i(x)$) a goal is defined: it can be a realistic or utopian value t_i. Then the goal is changed into two constraints stating $f_i(x) - d_i^- = t_i$, $d_i^- \geq 0$. We obtain a problem with more variables, where we have to minimize all the d_i; this problem is usually addressed by weighting factors or priorities, falling thus in one of the previous categories.

Note that all the methods we described (except the Weighting Objective Method) do not provide the whole efficient frontier (the set of non dominated points), but the only find one solution that will hopefully satisfy the user (or the Decision Maker). Also, *Interactive methods* (that try to interact with the Decision Maker to find a solution that he/she finds acceptable) have been described [109].

Various methods have been developed for the linear case [92] (where constraints are linear), but they are beyond the scope of this thesis. The interested reader can refer to [109].

It is worth noting that all these methods try to translate a MOP into one or more COPs; they do not address directly the MOP problem. Other methods address the MOP but are incomplete, exploiting genetic algorithms [32], tabu search [54].

26.5 Works Related to Partially-Ordered Constraint Optimization Problems

In other works, the concept of CSP with a partial ordering between solutions arises. The framework of Semiring based Constraint Satisfaction [10] generalizes various CSP extensions, like Fuzzy CSP [94], Possibilistic CSP [101], Partial CSP [36] and others. In Semiring based CSP (SCSP), each constraint is associated a level of satisfaction, given with a value belonging to a semiring. Propagation of constraints is defined with the operations in the semiring. Each solution can thus be associated a value in the semiring (considering the value obtained after complete propagation). Since over the semiring a partial ordering based on the addition operation ($A \preceq_S B \Leftrightarrow A + B = B$) can be defined, a partial ordering

is implicitly defined over the solution space. In this way, the MOP can be cast as an instance of the SCSP [11]. It is worth noting that the induced partial ordering is a lattice: given two elements A and B the Least Upper Bound is $A + B$ and the Greatest Lower Bound is $A \times B$. From this we can see that not every PCOP can be cast as an instance of a SCSP. The SCSP framework was also implemented as a programming language [49].

Another framework that subsumes many CSP extensions is the Valued CSP [102]. In this model, there is a valuation structure which is totally ordered. Each constraint is associated a value in the valuation structure denoting the impact of its valuation. It has been shown [9] that if one assumes a total order, then one can pass from any SCSP to an equivalent VCSP and vice versa, so the VCSP does not subsume the PCOP.

The Max-CSP [36], [113] deals with over-constrained problems. The aim is to find an assignment that violates as few constraints as possible; this can be done either by assigning a priority to each constraint or by ranking constraints hierarchically. The authors define a partial order among solutions defined by means of the violated constraints: an assignment A is better than B (in our symbolism, $B \preceq_m A$) if B violates (at least) all the constraints violated by A. The authors also provide a Branch-and-Bound algorithm based on the distance from the ideal solution (i.e., no constraints are violated). The Max-CSP can be simply converted into a MOP: we can turn some (or all) of the constraints into optimization functions: each function returns *one* when the corresponding constraint is satisfied and *zero* if it is violated. The obtained partial order is exactly the same as the one described in [36]. The algorithm presented here can thus be employed to find the assignments that violate a minimal set of constraints (in the $\preceq_m$ sense).

PART VII

Conclusions

27.0 Conclusions

In this thesis, we studied various problems related to artificial vision driven by constraints. We implemented a visual search system, with special attention to the generality of the proposed models and techniques, in order not to find sterile solutions to specific issues, but to provide general methodologies to solve problems in Artificial Intelligence with Constraint Programming tools, possibly starting fertile discussion and research. The techniques developed in the visual search system were applied to other types of problems, not directly related to artificial vision.

In the next sections, we summarize the main definitions and achievements, and we point out the applications to visual search and to other types of problems. We also propose some possible extensions and ideas that could be developed in future research.

In particular, in chapter 28 we discuss the work on Interactive Constraint Satisfaction; in chapter 29 we summarize the main achievements on Backtracking for Lazy domain evaluation. In 30 we give the conclusions and future work on the CLP language extension corresponding to the Interactive Constraint Satisfaction model. In chapter 31 we give report on the Partially-ordered Constraint Optimization model, and in 32 we give conclusions about 3D object modelling.

28.0 Interactive Constraint Satisfaction

An important source of inefficiency in Visual Search based on Constraints is the acquisition of visual features. The extraction of visual features is expensive, so the number of extracted features should be minimized: an image usually contains a huge number of features, so the attention of the system should be focussed on significant parts.

We have presented the Interactive Constraint Satisfaction framework, an extension of the CSP for dealing with unknown variable domains. The advantages of the proposed solution are several:

- we do not need to know all domain values in advance, before the computation starts;
- resulting domains are smaller than in standard CSPs and the efficiency of the constraint solver increases;
- the number of information extractions can be minimized;
- the data acquisition can be guided by interactive constraints. If they embed some form of locality, this leads to focus the attention of the producer of domain values;
- new values resulting from the acquisition can be inserted in variable domains and can be considered without restarting the constraint propagation process from scratch.

We also designed some ICSP solving algorithms based on active exploitation of constraints during search. We discussed how to measure the performance of an ICSP solving algorithm by considering the computational efforts of both the acquiring and solver modules, the communication cost and the interaction protocol. We produced experimental results on randomly generated CSPs showing that the ICSP has good performances considering many possible indexes (number of constraint checks, number of extracted elements, number of requests), unless the communication has a prevailing associated overhead.

The ICSP framework was developed for visual search problems. Constraint-based visual search can be subdivided into two phases: an acquisition phase and a matching phase. In the acquisition phase, the visual features are extracted from the image; in the matching phase, constraint satisfaction is used to associate image parts with the model.

Modelling the problem as an ICSP, and using ICSP solving algorithms gave good performance results, allowing to recognize objects in about 50% of the computation time required

by a non-interactive solver. The number of extracted features was typically halved, due manly to the focussing of the system to semantically significant image parts. Images usually contain huge number of visual features, thus it is important to avoid useless acquisition. Note that also in human vision the concept of attention is fundamental.

Apart from visual search, the ICSP model has been successfully applied to other types of problems with interaction. In many real-life problems, parameters are derived from the outer world. Often the acquisition of parameters is complex, because information has to be synthesized and encoded in symbolic form, or because they have to be provided by the user.

The ICSP model was applied to Planning [5], in this way, the planner retrieves only those values which are really needed for the plan construction.

The framework is very general, and other constraint reasoners can be considered as form of ICSP [73, 72].

The ICSP framework can be used when a constraint solver needs to acquire information about domains from the external world. An interesting point that could be studied in more detail is the acquisition through conjunction of constraints. We performed some tests with a data-base system, where domain values are stored in a data-base; data-bases are well suited for this type of investigation because the query language supports conjunction of conditions.

Other types of algorithms and solving strategies could be studied as well.

In the vision system, we will study the use of an ICSP model to improve also the *effectiveness* of the matching, not only the *efficiency*. For example, interactive constraints could inform the segmentation module of the inclination of a surface. The extraction of inclined surfaces is hard and often unreliable, but specialized methods can be provided. Different segmentation algorithms/parameters could be employed for the extraction of inclined surfaces.

Finally, we plan to extend the number of tests and study in detail which interactive algorithm performs better varying the parameters that represent the solving system. In other words, varying the relative performances of the two modules and the communication subnet, we plan to study which of the interactive algorithm is more promising.

29.0 Backtracking for Lazy Domain Evaluation

In order to deal efficiently with lazy domain evaluation, we need to record information about backtracking [23]; we proposed a non chronological backtracking rule needed when implementing lazy domain evaluation. We provided a meta-interpreter providing the required backtracking rule and an implementation of Minimal Forward Checking [23] exploiting it. The meta-interpreter allows an implementation of Minimal Forward Checking without extra-logic predicates. Some experimental results obtained with randomly generated problems have been shown.

In future work, we plan to implement a Prolog interpreter in a language not providing backtracking itself (e.g., Lisp). This could be important to show that a Prolog system providing the proposed predicate can be efficiently implemented. We believe that in this case, the introduced overhead would be minimal. In a Lisp implementation, we would not be forced to perform backtracking and redo part of the computation; we simply would not undo the necessary computation. Finally, we consider implementing it in a Warren Abstract Machine.

30.0 CLP(FD+$\mathcal{P}_{FD}$)

One of the main issues of the thesis was developing languages, techniques and frameworks for constraint satisfaction.

We developed a language, corresponding to the ICSP, that performs a constraint computation on variables with finite domains when information about domains is not fully known. Domains are channels of information, and are considered as first-class objects that can be themselves defined by means of constraints. The obtained language belongs to the CLP class and deals with two sorts: the FD sort on finite domains and the *S-Channel* sort for domains. We defined the concept of *known arc-consistency* and compared it with arc-consistency. We described a propagation engine for the FD sort exploiting known arc-consistency that provides the good efficiency level of ICSP [21] in a more general framework.

The framework can be generalized and consider, as a domain sort, many different sorts representing collections of values, as sets, streams or lists. In particular, the framework can be implemented on top of different systems. We presented the syntax and semantics of the language and we defined the specifications for the possible CLP($\mathcal{P}_{FD}$) languages that can be fruitfully exploited. Then, we have shown that two existing systems, Conjunto [50] and $\{log\}$ [25], with different expressive power and different efficiency levels, satisfy the defined specifications. Finally, we have shown some application examples.

Future work concerns implementation on top of different CLP languages, in order to study in more detail the problems and features which are inherited by the system in various environments. We also plan to extend the current implementation to exploit non-unary auxiliary constraints.

31.0 Partially-ordered Constraint Optimization Problems

In the classical Constraint Optimization Problem model, an objective function, representing, for example, a cost, is used to specify preferred solutions. In many real-life problems, however, feasible solutions are not sorted through a total order, but through a partial order. We defined the Partially-ordered Constraint Optimization Problem (PCOP) model, and provided an extension of Branch-and-Bound to find the set of solutions for which there is no feasible solution that is better in the partial order.

The concept of PCOP was developed for the visual search application, because the set of solutions are naturally ordered through a partial order. In 3D visual search, the model of the object can be given either in a view-centered model or in a object-centered model. In the first case, all the possible viewpoints must be described as an (I)CSP; in the second, the user provides a model of the object and the system preforms the matching of model parts with image parts. An *invisible* value is inserted in the domains, representing surfaces on the other side of the object. The *invisible* value satisfies the constraints by definition. The number of invisible surfaces in a consistent matching should be kept minimal through a partial order.

We have shown that, in the particular partial order used in visual search, a clever heuristics is enough to prove optimality. In this way, we can easily combine the efficiency provided by the ICSP model for interaction with the effectiveness of the recognition based on a PCOP.

The PCOP framework contains various types of real-life problems, like Multi-Objective Optimization. We presented an algorithm that extends Branch-and-Bound for Multi-Objective Optimization in CLP(FD). The algorithm is complete, meaning that it finds the whole non-dominated frontier. It is applicable to all the problems in CLP(FD), as it does not make any assumption on the structure of constraints. It uses Point Quad-Trees to efficiently store the previously found solutions. We compared the algorithm with a widely used method and it resulted more efficient, particularly in the difficult instances.

We now plan to apply the algorithm to over-constrained problems and problems with soft-constraints.

32.0 3D Object Modelling

We developed a Visual Search system first for 3D object recognition. We provided an Object-Centered model of 3D objects, employing *object constraints* to define the object and *visual constraints*, to discriminate possible and impossible views. We also applied the PCOP model in order to discard sub-optimal matchings. Both the PCOP model and *visual constraints* interface easily with the ICSP, proving the generality of the proposed frameworks.

In future work, we plan to employ Constraint Handling Rules [39] to impose visual constraints given the object model, and to apply machine learning techniques to infer visual constraints from examples.

APPENDIX A

APPENDIX A Proofs

Proposition A.0.1. *In algorithm 26.7, step 3 restricts the search to the first $b-1$ strips.*

Proof. The imposed constraint:

$$\neg\left(\overline{f(O_2)} \preceq_s \overline{f(X)}\right)$$

From the definition of $\preceq_s$ (eq. 26-1):

$$\neg\left(f_2(O_2) \leq f_2(X) \wedge \left\lceil \frac{f_1(O_2)}{f_1(O_2)} b \right\rceil \leq \left\lceil \frac{f_1(X)}{f_1(O_2)} b \right\rceil\right)$$

$$\neg\left(true \wedge b \leq \left\lceil \frac{f_1(X)}{f_1(O_2)} b \right\rceil\right)$$

$$b > \left\lceil \frac{f_1(X)}{f_1(O_2)} b \right\rceil$$

so the strip that contains X should be strictly before b. □

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Hal Abelson, Gerald J. Sussman, and Julie Sussman, *Structure and interpretation of computer programs*, 6 ed., MIT Press, USA, 1985.

[2] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve, *ECL^{i}PS^e user manual, release 5.2*, IC-Parc, Imperial College, London, UK, 2001.

[3] Fahiem Bacchus and Adam J. Grove, *On the Forward Checking Algorithm*, Proceedings First International Conference on Constraint Programming (Cassis, France) (Ugo Montanari and Francesca Rossi, eds.), Lecture Notes in Computer Science, vol. 976, Springer-Verlag, September 19–22 1995, pp. 292–309.

[4] Fahiem Bacchus and Peter van Beek, *On the conversion between non-binary and binary constraint satisfaction problems*, National Conference on Artifical Intelligence (AAAI-98) (Madison, Wisconsin, USA), AAAI Press / The MIT Press, July 26-30 1998, pp. 310–318.

[5] Rosy Barruffi, Evelina Lamma, Paola Mello, and Michela Milano, *Least commitment on variable binding in presence of incomplete knowledge*, Proceedings of the European Conference on Planning (ECP99) (Durham, UK) (S. Biundo and M. Fox, eds.), Lecture Notes in Computer Science, vol. 1809, Springer, September 8 - 10 1999, pp. 159–171.

[6] Christian Bessiére, *Arc-consistency and arc-consistency again*, Artificial Intelligence **65** (1994), no. 1, 179–190.

[7] Christian Bessiére, Eugene C. Freuder, and Jean-Charles Régin, *Using constraint metaknowledge to reduce arc consistency computation*, Artificial Intelligence **107** (1999), no. 1, 125–148.

[8] Irving Biederman, *Human image understanding: recent research and theory*, Computer Vision, Graphics and Image Processing **32** (1985), 29–73.

[9] Stefano Bistarelli, Hélène Fargier, Ugo Montanari, Francesca Rossi, Thomas Schiex, and Gérard Verfaillie, *Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison*, Over-Constrained Systems (Selected papers from the Workshop on Over-Constrained Systems at CP'95) (M. Jampel, E. Freuder, and M. Maher, eds.), Lecture Notes in Computer Science, vol. 1106, Springer-Verlag, 1996, pp. 111–150.

[10] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi, *Constraint solving over semirings*, Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (San Mateo) (Chris S. Mellish, ed.), Morgan Kaufmann, August 20–25 1995, pp. 624–630.

[11] ———, *SCLP Semantics for (Multi-Criteria) Shortest Path Problems*, Proceedings of CP-AI-OR'99 Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (Ferrara, Italy), February 25–26 1999.

[12] Michael Boshra and Hong Zhang, *A constraint-satisfaction approach for 3D vision/ touch-based object recognition*, IEEE/RSJ International Conference on Intelligent Robots and Systems, vol. 3, IEEE, 1995, pp. 368–373.

[13] Maurice Bruynooghe and Luís Moniz Pereira, *Deduction revision by intelligent backtracking*, Implementations of Prolog (J. A. Campbell, ed.), Ellis Horwood, Chichester, 1984, pp. 196–215.

[14] Mats Carlsson, Johan Widén, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland, *SICStus prolog user's manual*, Tech. Report T91:15, Swedish Institute of Computer Science, June 1995.

[15] Ilyas Cicekli, *An intelligent backtracking schema in a logic programming environment*, ACM SIGPLAN Notices **32(3)** (1997), 42–49.

[16] Philippe Codognet and Daniel Diaz, *Compiling constraints in* `clp(`*FD*`)`, Journal of Logic Programming **27** (1996), no. 3, 185–226.

[17] Jacques Cohen, Pascal Koiran, and Catherine Perrin, *Meta-level interpretation of CLP(Lists)*, Constraint Logic Programming - Selected Research (F. Benhamou and A. Colmerauer, eds.), MIT Press, 1993, pp. 457–481.

[18] Philip T. Cox and Tomasz Pietrzykowski, *Deduction Plans: A Basis for Intelligent Backtracking*, IEEE Transactions on Pattern Analysis and Machine Intelligence **3** (1981), no. 1, 52–65.

[19] Rita Cucchiara, Marco Gavanelli, Evelina Lamma, Paola Mello, Michela Milano, and Massimo Piccardi, *Extending CLP(FD) with interactive data acquisition for 3D visual object recognition*, Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming (London), Practical Application Company, April 19–21 1999, pp. 137–155.

[20] ———, *Constraint propagation and value acquisition: why we should do it interactively*, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (Stockholm, Sweden) (Thomas Dean, ed.), Morgan Kaufmann, July 31 – August 6 1999, pp. 468–477.

[21] ———, *From eager to lazy constrained data acquisition: A general framework*, New Generation Computing **19** (2001), no. 4, 339–367.

[22] Rina Dechter and Avi Dechter, *Belief maintenance in dynamic constraint networks*, Proceedings of the 7th National Conference on Artificial Intelligence (St. Paul, MN) (Tom M. Smith, Reid G.; Mitchell, ed.), Morgan Kaufmann, August 1988, pp. 37–42.

[23] Michael J. Dent and Robert E. Mercer, *Minimal forward checking*, Proceedings of the Sixth International Conference on Tools with Artificial Intelligence, 1994, pp. 432–438.

[24] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier, *The constraint logic programming language CHIP*, Proceedings of the International Conference on Fifth Generation Computer System (Tokyo, Japan), OHMSHA Ltd. Tokyo and Springer-Verlag, November 28–December 2 1988, pp. 693–702.

[25] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi, {*log*}*: A language for programming in logic with finite sets*, Journal of Logic Programming **28(1)** (1996), 1–44.

[26] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi, *Sets and constraint logic programming*, ACM Transactions on Programming Languages and Systems **22** (2000), no. 5, 861–931.

[27] Agostino Dovier and Gianfranco Rossi, *Embedding extensional finite sets in CLP*, Proceedings of the 1993 International Symposium on Logic Programming (British Columbia, Canada), MIT Press, October26–29 1993, pp. 540–556.

[28] Bruce A. Draper, Allen R. Hanson, and Edward Riseman, *Knowledge-directed vision: control, learning and integration*, Proc. of IEEE, vol. 84, n. 11, 1996, pp. 1625–1681.

[29] Sahibsingh A. Dudani, Kenneth J. Breeding, and Robert B. McGhee, *Aircraft identification by moment invariants*, IEEE Transactions on Computers **26** (1977), no. 1, 39–46.

[30] David W. Eggert, Kevin W. Bowyer, Charles R. Dyer, Henrik I. Christensen, and Dmitry B. Goldgof, *The scale space aspect graph*, IEEE Transactions on Pattern Analysis and Machine Intelligence **15** (1993), 1114–1130.

[31] Raphael A. Finkel and Jon Louis Bentley, *Quad trees: A data structure for retrieval on composite keys*, Acta Informatica **4** (1974), no. 1, 1–9.

[32] Carlos M. Fonseca and Peter J. Fleming, *An Overview of Evolutionary Algorithms in Multiobjective Optimization*, Evolutionary Computation **3** (1995), no. 1, 1–16.

[33] Eugene C. Freuder, *Synthesizing constraint expressions*, Communication of the ACM **21** (1978), no. 11, 958–966.

[34] ———, *A sufficient condition for backtrack free search*, Communication of the ACM **29** (1982), no. 1, 24–32.

[35] ———, *In pursuit of the holy grail*, ACM Computing Surveys **28** (1996), no. 4es, 63.

[36] Eugene C. Freuder and Richard J. Wallace, *Partial constraint satisfaction*, Artificial Intelligence **58** (1992), no. 1-3, 21–70.

[37] ———, *Suggestion strategies for constraint-based matchmaker agents*, Principles and Practice of Constraint Programming - CP98, 4th International Conference, Proceedings (Pisa, Italy) (Michael J. Maher and Jean-Francois Puget, eds.), Lecture Notes in Computer Science, vol. 1520, Springer, October 26 – 30 1998, pp. 192–204.

[38] Markus PJ Fromherz and John Conley, *Issues in reactive constraint solving*, Proceedings of COTIC'97 - Workshop in CP'97 (Linz, Austria), November 1997.

[39] Thom Frühwirth, *Constraint Handling Rules*, Constraint Programming: Basic and Trends. Selected Papers of the 22nd Spring School in Theoretical Computer Sciences (Andreas Podelski, ed.), Lecture Notes in Computer Science, vol. 910, Springer-Verlag, Châtillon-sur-Seine, France, May 1994, pp. 90–107.

[40] ———, *Theory and practice of constraint handling rules, special issue on constraint logic programming*, Journal of Logic Programming **37** (1998), no. 1-3, 95–138.

[41] John Gaschnig, *A general backtrack algorithm that eliminates most redundant tests*, Proceedings of the 5th International Joint Conference on Artificial Intelligence (Cambridge, MA) (Raj Reddy, ed.), William Kaufmann, August 1977, pp. 457–457.

[42] ———, *Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignments problems*, Proceedings of the Canadian Artificial Intelligence Conference, 1978.

[43] Marco Gavanelli, *Partially ordered constraint optimization problems*, Principles and Practice of Constraint Programming, 7th International Conference - CP 2001 (Paphos, Cyprus) (Toby Walsh, ed.), Lecture Notes in Computer Science, vol. 2239, Springer Verlag, November 26 – December 1 2001, p. 763.

[44] Marco Gavanelli, Evelina Lamma, Paola Mello, and Michela Milano, *Domains as first class objects in CLP(FD)*, APPIA–GULP–PRODE '99 Joint Conference on Declarative Programming - Proceedings (L'Aquila, Italy) (M.C. Meo and M. Vilares Ferro, eds.), September 6–9 1999, pp. 411–424.

[45] ———, *Domains as first class objects in CLP(FD)*, Proceedings of the International Conference on Logic Programming ICLP'99 (Las Cruces, N.M.) (Danny De Schreye, ed.), MIT Press, November 29 – December 4 1999, p. 608.

[46] ———, *Performance measurement of interactive CSP search algorithms*, AI*IA Notizie **XIII** (2000), no. 1, 48–51.

[47] ———, *Exploiting constraints for domain managing in CLP(FD)*, 4th International Workshop on Frontiers of Combining Systems - FroCoS'2002 (Santa Margherita Ligure, Italy) (A. Armando, ed.), Lecture Notes in Artificial Intelligence, Springer Verlag, April 8-10 2002, To appear.

[48] Marco Gavanelli and Michela Milano, *On the need for a different backtracking rule when dealing with late evaluation*, Electronic Notes in Theoretical Computer Science **30** (1999), no. 2.

[49] Y. Georget and P. Codognet, *Compiling semiring-based constraints with clp(FD,S)*, Principles and Practice of Constraint Programming - CP98, 4th International Conference (Pisa, Italy) (Michael J. Maher and Jean-Francois Puget, eds.), Lecture Notes in Computer Science, vol. 1520, Springer, October 1998, pp. 205–219.

[50] Carmen Gervet, *Propagation to reason about sets: Definition and implementation of a practical language*, Constraints **1** (1997), 191–244.

[51] Carmen Gervet, Yves Caseau, and Denis Montaut, *On refining ill-defined constraint problems: A case study in iterative prototyping*, PACLP-99 (London), 1999, pp. 255–275.

[52] Fred Glover, *Heuristics for integer programming using surrogate constraints*, Decision Sciences **8** (1977), 156–166.

[53] Solomon W. Golomb and Leonard D. Baumert, *Backtrack programming*, Journal of the ACM **12** (1965), no. 4, 516–524.

[54] Michael Pilegaard Hansen, *Tabu search for multiobjective optimization: MOTS*, 13th International Conference on Multiple Criteria Decision Making (MCDM'97) (Cape Town, South Africa), January 1997.

[55] Robert M. Haralick and Gordon L. Elliott, *Increasing tree search efficiency for constraint satisfaction problems*, Artificial Intelligence **14** (1980), no. 3, 263–313.

[56] Robert M. Haralick and Linda G. Shapiro, *Computer and robot vision*, vol. 1, Addison-Wesley Publishing Company, Inc., Reading, MA, 1992.

[57] Peter Henderson and James H. Morris, *A lazy evaluator*, 3rd ACM symposium on Principles of Programming Languages, 1976, pp. 90–103.

[58] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach - 2nd edition*, Morgan Kaufmann, San Mateo, CA, 1996.

[59] Pascal Van Hentenryck, *Constraint satisfaction in Logic Programming*, MIT Press, 1989.

[60] Pascal Van Hentenryck and Yves Deville, *The cardinality operator: A new logical connective for constraint logic programming*, Eighth International Conference on Logic Programming (ICLP-8) (Paris, France) (K. Furukawa, ed.), MIT Press, June 24–28 1991, pp. 745–759.

[61] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng, *A generic arc-consistency algorithm and its specializations*, Artificial Intelligence **57** (1992), 291–321.

[62] John H. Holland, *Adaption in natural and artificial systems*, University of Michigan Press, Ann Arbor, 1975.

[63] Adam Hoover, Gillian Jean-Baptiste, Xiaoyi Jiang, Patrick J. Flynn, Horst Bunke, Dmitry B. Goldgof, Kevin Bowyer, David W. Eggerf, Andrew Fitzgibbon, and Robert Fisher, *An experimental comparison of range image segmentation algorithms*, IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) **18** (1996), no. 7, 673–689.

[64] Ming-Kuei HU, *Visual pattern recognition by moment invariants*, IEEE Transactions on Information Theory **8** (1962), 179–187.

[65] David A. Huffman, *Impossible objects as nonsense sentences*, Machine Intelligence 6 (Edinburgh) (Bernard Meltzer and Donald Michie, eds.), Edinburgh University Press, 1971, pp. 295–323.

[66] Yuji Ijiri, *Management goals and accounting for control*, North Holland, Chicago, 1965.

[67] Joxan Jaffar and Jean-Louis Lassez, *Constraint logic programming*, Conference Record 14th Annual ACM Symposium on Principles of Programming Languages (Munich, Germany), ACM SIGACT/SIGPLAN, January 21–23 1987, pp. 111–119.

[68] Joxan Jaffar and Michael J. Maher, *Constraint logic programming: a survey*, Journal of Logic Programming **19-20** (1994), 503–582.

[69] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey, *The semantics of constraint logic programs*, Journal of Logic Programming **37(1-3)** (1998), 1–46.

[70] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H.C. Yap, *The CLP(R) language and system*, ACM Transaction on Programming Language and Systems **14** (1992), no. 3, 339–395.

[71] Scott Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi, *Optimization by simmulated annealing*, Science **220** (1983), no. 4598, 671–680.

[72] Craig A. Knoblock, José Luis Ambite, Steven Minton, Cyrus Shahabi, Mohammad Kolahdouzan, Maria Muslea, Jean Oh, and Snehal Thakkar, *Integrating the world: The WorldInfo assistant*, International Conference on Artificial Intelligence (IC-AI) (Las Vegas, Nevada, USA), CSREA Press, June 25 – 28 2001.

[73] Craig A. Knoblock, Steven Minton, José Luis Ambite, Maria Muslea, Jean Oh, and Martin Frank, *Mixed-initiative, multi-source information assistants*, Tenth International World Wide Web Conference (Hong Kong, China), ACM, May 1 – 5 2001, pp. 697–707.

[74] Jan J. Koenderink and Ans J. van Doorn, *The internal representation of solid shape with respect to vision*, Biological Cybernetics **32** (1979), no. 4, 211–216.

[75] Thomas Kolbe, Lutz Plümer, and Armin B. Cremers, *Using constraints for the identification of buildings in aerial images*, 2nd International Conference on the Practical Applications of Constraint Programming - PACT'96 (London, U.K.), 1996, pp. 143–154.

[76] Robert Kowalski, *Logic for problem solving*, North-Holland, New York, amsterdam, oxford, 1979.

[77] Vipin Kumar and Yow-Jian Lin, *A data-dependency-based intelligent backtracking scheme for Prolog*, Journal of Logic Programming **5** (1988), no. 2, 165–181.

[78] E.L. Lawler and D.E. Wood, *Branch-and-bound methods: a survey*, Operations Research **14(4)** (1966), 699–719.

[79] Bruno Legeard and Emmanuel Legros, *Short overview of the CLPS system*, Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91 (Passau, Germany) (J. Małuszyński and M. Wirsing, eds.), Lecture Notes in Computer Science, Springer-Verlag, August 1991, pp. 431–433.

[80] Alan K. Mackworth, *Consistency in networks of relations*, Artificial Intelligence **8** (1977), 99–118.

[81] Daniel Mailharro, *A classification and constraint-based framework for configuration*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing **12** (1998), 383–397.

[82] Silvano Martello and Paolo Toth, *Knapsack problems: algorithms and computer implementations*, John Wiley & Sons, 1990.

[83] Sanjay Mittal and Brian Falkenhainer, *Dynamic constraint satisfaction problems*, Proceedings of the 8th National Conference on Artificial Intelligence, AAAI-90 (Boston, Massachusetts), AAAI Press / The MIT Press, July 29–August 3 1990, pp. 25–32.

[84] Roger Mohr and Thomas C. Henderson, *Arc and path consistency revisited*, Artificial Intelligence **28** (1986), no. 2, 225–233.

[85] Ugo Montanari, *Networks of constraints: Fundamental properties and applications to picture processing*, Information Sciences **7** (1974), 95–132.

[86] Jan A. Mulder, Alan K. Mackworth, and William S. Havens, *Knowledge structuring and constraint satisfaction: The mapsee approach*, IEEE Transactions on Pattern Analysis and Machine Intelligence **10** (1988), no. 6, 866–879.

[87] Eric Persoon and King-Sun Fu, *Shape discrimination using Fourier descriptors*, IEEE Transactions on Pattern Analysis and Machine Intelligence **8** (1986), no. 3, 388–391.

[88] Jean Ponce, David Chelberg, and Wallace B. Mann, *Invariant properties of straight homogeneous generalized cylinders and their contours*, IEEE Transactions on Pattern Analysis and Machine Intelligence **11** (1989), no. 9, 951–966.

[89] Steven Prestwich, *Three implementations of branch-and-bound in CLP*, Proceedings of Fourth Compulog-Net Workshop on Parallelism and Implementation Technologies (Bonn), September 1996.

[90] Jean-Francois Puget, *On the satisfiability of symmetrical constrained satisfaction problems*, Proceedings of ISMIS'93 (J. Komorowski and Z.W. Ras, eds.), Lecture Notes in Artificial Intelligence, vol. 689, Springer-Verlag, 1993, pp. 350–361.

[91] ———, *A C++ implementation of CLP*, Tech. Report 94-01, ILOG Headquarters, 1994.

[92] Meir J. Rosenblatt and Zilla Sinuany-Stern, *Generating the discrete efficient frontier to the capital budgeting problem*, Operations Research **37** (1989), no. 3, 384 – 394.

[93] Peter Van Roy and Seif Haridi, *Mozart: A programming system for agent applications*, International Workshop on Distributed and Internet Programming with Logic and Constraint Languages, November 1999, Part of International Conference on Logic Programming (ICLP 99).

[94] Zsófia Ruttkay, *Fuzzy constraint satisfaction*, Proceedings 1st IEEE Conference on Evolutionary Computing (Orlando), 1994, pp. 542–547.

[95] Daniel Sabin and Eugene C. Freuder, *Contradicting Conventional Wisdom in Constraint Satisfaction*, Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94, Rosario, Orcas Island, Washington, USA (Alan Borning, ed.), Lecture Notes in Computer Science, vol. 874, May 1994, pp. 10–20.

[96] Mindia E. Salukvadze, *On the existence of solution in problems of optimization under vector valued criteria*, Journal of Optimization Theory and Applications **12** (1974), no. 2, 203–217.

[97] Hanan Samet, *Deletion in two-dimensional quad trees*, Communications of the ACM **23** (1980), no. 12, 703–710.

[98] ———, *The quadtree and related hierarchical data structures*, ACM Computing Surveys **16** (1984), no. 2, 187–260.

[99] ———, *The design and analysis of spatial data structures*, Addison-Wesley, Reading, MA, 1990.

[100] Vijay A. Saraswat, *Concurrent constraint programming*, MIT Press, 1993.

[101] Thomas Schiex, *Possibilistic constraint satisfaction problems or "How to handle soft constraints?"*, Proceedings of the 8th Conference on Uncertainty in Artificial Intelligence (San Mateo, CA, USA) (Dubois, Didier, Wellman, Michael P., D'Ambrosio, Bruce, and Phillipe Smets, eds.), Morgan Kaufmann Publishers, July 1992, pp. 268–275.

[102] Thomas Schiex, Hélène Fargier, and Gérard Verfaillie, *Valued constraint satisfaction problems: Hard and easy problems*, Proceedings of the Fourteenth International Joint

Conference on Artificial Intelligence (San Mateo) (Chris S. Mellish, ed.), Morgan Kaufmann, August 20–25 1995, pp. 631–639.

[103] Thomas Schiex, Jean-Charles Régin, Christine Gaspin, and Gérard Verfaillie, *Lazy arc consistency*, Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference (Menlo Park), AAAI Press / MIT Press, August 4–8 1996, pp. 216–221.

[104] Thomas Schiex and Gérard Verfaillie, *Nogood recording for static and dynamic constraint satisfaction problems*, Proceedings of the 5th International Conference on Tools with Artificial Intelligence (Los Alamitos, CA, USA), IEEE Computer Society Press, November 1993, pp. 48–55.

[105] Marek Sergot, *A query-the-user facility for logic programming*, Integrated Interactive Computing Systems (P. Degano and E. Sandewall, eds.), North-Holland, 1983, pp. 27–41.

[106] Ehud Y. Shapiro (ed.), *Concurrent prolog - vol. i*, MIT Press, 1987.

[107] ———, *The family of concurrent logic programming languages*, ACM Computing Surveys **21** (1989), no. 4, 413–510.

[108] Barbara M. Smith and Martin E. Dyer, *Locating the phase transition in contraint satisfaction problems*, Artificial Intelligence **81** (1996), no. 1-2, 155–181.

[109] Ralph E. Steuer, *Multiple criteria optimization: Theory, computation, and application*, Wiley, New York, 1986.

[110] D. Terzopoulos and D. Metaxas, *Dynamic 3d models with local and global deformations: Deformable superquadrics*, IEEE Transactions on Pattern Analysis and Machine Intelligence **13** (1991), no. 7, 703–714.

[111] Edward P.K. Tsang, *Foundation of constraint satisfaction*, Academic Press, 1993.

[112] David Vernon, *Machine vision: Automated visual inspection and robot vision*, Prentice Hall, New York, 1991.

[113] Richard J. Wallace, *Enhancements of branch and bound methods for the maximal constraint satisfaction problem*, Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference (Menlo Park), AAAI Press / MIT Press, August 4–8 1996, pp. 188–195.

[114] Richard J. Wallace and Eugene C. Freuder, *Ordering heuristics for arc consistency algorithms*, Proceedings of the Ninth Canadian Conference on Artificial Intelligence (Vancouver, BC), Morgan Kaufmann, May 1992.

[115] David L. Waltz, *Generating semantic descriptions from drawings of scenes with shadows*, Tech. Report AI-TR-271, A.I. Lab., M.I.T., Cambridge, Mass., 1972.

[116] ———, *Generating semantic descriptions from drawings of scenes with shadows*, The Psychology of Computer Vision (P.H. Winston, ed.), McGraw-Hill, New York, 1975.

[117] Luc N. Van Wassenhove and Ludo F. Gelders, *Solving a bicriterion scheduling problem*, European Journal of Operational Research **4** (1980), no. 1, 42–48.

[118] Daniel S. Weld, *An introduction to least commitment planning*, AI Magazine **15** (1994), no. 4, 27–61.

[119] Ming-Hsuan Yang and Michael M. Marefat, *Constraint-based feature recognition: handling non-uniqueness in feature interactions*, Proceedings of IEEE International Conference on Robotics and Automation (Minneapolis, MN, USA), IEEE, April 1996, pp. 1505–1510.

[120] Eckart Zitzler and Lothar Thiele, *Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach*, IEEE Transactions on Evolutionary Computation **3** (1999), no. 4, 257–271.

[121] Monte Zweben and Megan Eskey, *Constraint satisfaction with delayed evaluation*, Proceedings of the 11th International Joint Conference on Artificial Intelligence (Detroit, MI, USA) (N.S. Sridharan, ed.), Morgan Kaufmann, August 20–25 1989, pp. 875–880.